# APT: A principled design for an animated view of program execution for novice programmers

## Thesis



# APT: A Principled Design for an Animated View of Program Execution for Novice Programmers

**Tim Rajan**

Thesis submitted in partial fulfillment of requirements for Ph.D in Psychology, 14th November 1986

**Human Cognition Research Laboratory**
**The Open University**
**Milton Keynes MK7 6AA**
**U.K**

EX35

HIGHER DEGREES OFFICE

LIBRARY AUTHORISATION

STUDENT: TIMOTHY M. RAJAN SERIAL NO: HDM66028

DEGREE: DOCTOR OF PHILOSOPHY

TITLE OF THESIS: ART: THE DESIGN OF ANIMATED TRACING TOOLS FOR NOVICE PROGRAMMERS

I confirm that I am willing that my thesis be made available to readers and may be photocopied, subject to the discretion of the Librarian.

Signed: T M Rajan Date: 19.11.86

## ABSTRACT

This thesis is concerned with the principled design of a computational environment which depicts an animated view of program execution for novice programmers. We assert that a principled animated view of program execution should benefit novice programmers by: (i) helping students conceptualize what is happening when programs are executed; (ii) simplifying debugging through the presentation of bugs in a manner which the novice will understand; (iii) reducing program development time.

The design is based on principles which have been extracted from three areas: (i) the problems that novices encounter when learning a programming language; (ii) the general design principles for computer systems; and (iii) systems which present a view of program execution.

The design principles have been embodied in three 'canned' stepper displays for Prolog, Lisp and 6502 Assembler. These prototypes, called APT-0 (Animated Program Tracer), demonstrate that the design principles can be broadly applied to procedural and declarative; low and high level languages. Protocol data was collected from subjects using the prototypes in order to check the direction of the research and to suggest improvements in the design. These improvements have been incorporated in a real implementation of APT for Prolog.

This principled approach embodied by APT provides two important facilities which have previously not been available, firstly a means of demonstrating dynamic programming concepts such as variable binding, recursion, and backtracking, and secondly a debugging tool which allows novices to step through their own code watching the virtual machine in action. This moves towards simplifying the novice's debugging environment by supplying program execution information in a form that the novice can easily assimilate.

An experiment into the misconceptions novices hold concerning the execution of Prolog programs shows that the order of database search, and the concepts of variable binding, unification and backtracking are poorly understood. A further experiment was conducted which looked at the effect that APT had on the ability of novice Prolog programmers to understand the execution of Prolog programs. This demonstrated that the performance of subjects significantly increased after being shown demonstrations of the execution of Prolog programs on APT, while the control group who saw no demonstration showed no improvement.

The experimental evidence demonstrates the potential of APT, and the principled approach which it embodies, to communicate run-time information to novice programmers, increasing their understanding of the dynamic aspects of the Prolog interpreter.

APT, uses an object centred representation, is built on top of a Prolog interpreter and environment, and is implemented in Common Lisp and Zeta Lisp and runs on the Symbolics™ 3600 range of machines.

## DEDICATION

This thesis is dedicated to all my family for their love and support, and to Jo for her enlightenment.

## ACKNOWLEDGEMENTS

I would like to thank the following people for their time and help during the time I have spent studying for my PhD.

Marc Eisenstadt for his interest, motivation, close and friendly supervision, which has made the past three years one of the most enjoyable experiences of my life.

Tim O'Shea for his original co-supervision, and his always useful comments.

I would especially like to thank Marc and Tim for their support at a crucial time at the start of my PhD for which I am very grateful, and without which none of this would have happened.

Martin Levoi for his programs which allowed me to build and run both the APT prototypes, and the associated experiments.

Tony Hasemer for his work on the Prolog interpreter and numerous comments both throughout the PhD and on previous drafts of this thesis.

Enrico Motta for his helpful introduction to the Symbolics, and many hours of discussion and argument.

Doreen Warwick for her invaluable help in organising experiments at a moments notice. Certain experiments would not have been carried out in time without her knowledge of the OU system, and her persausive ways.

D309 students of '86 for their time as experimental subjects when they had plenty of work to do for their course prior to exams.

The Psychology Department for making the environment at work so much fun.

Special thanks go to the subjects who took the trouble to come to Walton Hall for extensive experiments, namely Dianne Parker; Richard Pearce; Trevor Jones; Valerie Crowley; Rita Marshall; Ruby Peard; Christine Rawlings; Don Mahone; Lynda Couzens; Niel Parish; Martin Derby, and Tim Guy.

# TABLE OF CONTENTS

## CHAPTER 1 INTRODUCTION



## CHAPTER 2 RELATED RESEARCH

## CHAPTER 3 DESIGN PRINCIPLES



## CHAPTER 4 THE PROTOTYPES OF APT



## CHAPTER 5 EVALUATION OF THE PROTOTYPES



## CHAPTER 6 APT

## CHAPTER 7 THE EFFECT OF APT ON NOVICES



## CHAPTER 8 DISCUSSION

## FIGURES

CHAPTER 4



CHAPTER 5

CHAPTER 6



CHAPTER 7

# CHAPTER 1
# INTRODUCTION

## 1.1 The Problem

In most programming courses novices are introduced to dynamic concepts as abstract theories, or by analogy with non-programming ideas. Some concepts such as the stack, the linked list, and arrays are usually taught with the aid of static graphic tools, for example matrices, lists and pointers. However, the programming concepts which have a dynamic nature i.e. recursion, iteration, variable binding, flow of control, parameter passing and search methods are more difficult to represent using static graphics, or even 'snapshot sequences', as their action is not easily conveyed with fixed images. The shortcomings of using static representations of recursion and backtracking as a teaching aid can be seen in many books on programming languages (Touretzky, 1984; Hasemer, 1984; Clocksin and Mellish, 1981; Winston and Horn, 1981, 1984). There is a body of research showing that novices have problems learning the dynamic concepts mentioned above. (Sime, Green and Guest, 1973; Soloway, Bonar and Ehrlich, 1983; Waters, 1979; Brooks, 1977; Kahney, 1982; Anderson, Farrel and Sauers, 1982; Eisenstadt, Breuker and Evertsz, 1984).

One reason that novices find difficulty in learning these programming concepts is the problem of associating the abstract nature of the description of the concept to the structure and content of the programs they are attempting to write. For example, recursion may be described as the referencing of a function/rule from within that

function/rule, and presented as the execution of fresh copies of the function/rule. The problem the novice has now is to map that description of recursion onto the program s/he is writing, in order to understand what the program is doing, or what it is supposed to do. This problem will occur whether it concerns recursion, parameter passing, variable binding or flow of control.

It is a large step to take from the theory of programming to the practice of writing programs. In fact it is a Catch-22, as the novice needs to understand these dynamic concepts before s/he can write useful working programs, but on the other hand s/he will need some experience of such concepts in simple working programs, before s/he can build a mental model of program execution based upon concrete examples. Even the tools available in programming environments to view program execution only present the user with a static or snapshot representation of a program's execution, and so fall short of providing a concrete base on which the novice may build a model of program execution.

Most existing tracers and single steppers do not explicitly show the surface evaluation of the code the user has written (surface evaluation means the evaluation of the user's program in terms of the code that program is written in) but a mixture of program output, the internal evaluation of the code, and the code the user has written, generally in a format that is some variant of the internal representation of the user's code. Once again this presents the novice with the problem of associating the trace with the program s/he has written. How can the novice understand the output of a program trace when it bears no resemblance to the program s/he has written? Catch-22 again! The user needs to have a good model of program execution before s/he can have a chance of understanding the program trace. So these traditional tracers provide no help to the novice who is trying to learn a programming language, with its myriad concepts. In fact it is just an extra cognitive burden for the novice. To illustrate the dilemma we will consider a simple Prolog program and the output of that program

when run through a traditional tracer, but before doing this I will present a brief summary of Prolog.

Prolog uses a modified predicate logic notation to express facts, rules of inference, and queries. The following program contains three facts and one rule:

```
kisses(mary,john).
kisses(john,june).

has_flu(X):-
     kisses(Y,X),
     has_flu(Y).
has_flu(mary).
```

The program has the following intuitive meaning. The first two facts state that 'mary kisses john', and that 'john kisses june'. The rule has both a declarative and a procedural meaning. Declaratively, the rule states 'for all X, if Y kisses X and Y has flu, then X has flu'. Procedurally, the rule means 'to prove that somebody has flu, (1) show that that person kisses someone, and also (2) shown that someone has the flu . Finally the last fact states that 'mary has flu'. These rules and facts together form the Prolog database.

All of the symbols inside the brackets are arbitrary tokens, so that the last fact, for example, specifies 'has_flu' to be a unary predicate. Upper case letters, or words beginning with an upper case letter denote a variable, while those in lower case denote constants.

Queries are addressed to the Prolog interpreter as follows:

```
?- kisses(mary,john).

?- kisses(mary,X).
```

The first query asks, 'is it true that mary kisses john', and the second query, 'is it true that mary kisses anyone'. In both cases the interpreter searches sequentially for facts or rules which can be used to deduce the truth of the query expression. In the latter case, the interpreter also shows the 'binding' (pattern match) of the variable 'X' to the first possible item it matches against in the database. Further matches are

produced by the interpreter if requested by the user, who types a semicolon to the 'More?' prompt which appears after the variable binding/s.

Here is what happens if we run the above program using the standard 'spy' trace package:

```
?- has_flu(june).
** (1) Call : has_flu(june)?
** (2) Call : kisses(_1, june)?
** (2) Exit : kisses(john, june)?
** (3) Call : has_flu(john)?
** (4) Call : kisses(_2, john)?
** (4) Exit : kisses(mary, john)?
** (5) Call : has_flu(mary)?
** (6) Call : kisses(_3, mary)?
** (6) Fail : kisses(_3, mary)?
** (5) Exit : has_flu(mary)?
** (3) Exit : has_flu(john)?
** (1) Exit : has_flu(june)?
yes
?-
```

Snapshots from the corresponding APT-0 display for this can be seen in an abbreviated form in chapter 4.

The main problem with the above stepped output is that it is difficult to relate to the original source code. This is because the stepped output is presented in a very different format to the Prolog program, making it difficult to associate the dynamic events occuring in the trace with the static representation of the Prolog program.

The trace leaves the following important questions unanswered:

1) How are the variables instantiated; where do they get their values from?

2) How and why does the program finish?

3) How does the flow of control work?

4) How does the matching of clauses work, and in what order?

This Prolog trace is based on the 'Byrd' box model of Prolog execution (Byrd, 1980). To use this tool the novice must fully understand this model of program execution to get any information from the stepper. The 'Byrd' box model displays

what has happened at certain important points during program execution, but not how it got to these points which is most useful to a beginner.

Remember that the above example is showing a correct and simple program: just think how confusing a more complex program would look, or even a buggy one.

The motivation for this work has come from a programming course for novices which is part of the Cognitive Psychology course at the Open University. The original language used on this course, SOLO, was an in-house package written specially for novices, including a very friendly environment and manual (Eisenstadt, 1983). The research projects which evaluated SOLO's effectiveness as a novice programming language found that the major problems novices appeared to have with SOLO were understanding dynamic concepts i.e. variable binding, control flow, recursion and iteration, argument passing (Lewis, 1980; Kahney, 1982). Similar problems are encountered by novices learning other programming languages, as discussed in Chapter 2.

## 1.2 The Approach

Progress in a field often follows the development of a more powerful method of representing the ideas within that field. For example advances in program and text editing have followed the development of interactive user aids. This has led to many improvements, the most commonplace of which is the screen editor in word processing. The screen editor, a vast improvement on the previous line editors, allows the material being worked on to be viewed in meaningful units as it is in everyday life, for instance paragraphs and pages, rather than a series of single lines.

Novice users of a programming language could benefit from an improvement in the way that the execution of programs written in that language are presented to them in the environment they use. Programming languages are generally taught using static methods (paper and pencil) when many of the concepts and techniques involved are

dynamic in nature. These concepts/techniques should be easier to understand and learn if they are presented in a dynamic fashion and in terms of the programming code the user has written.

If the ideas used in screen editing are transferred to other parts of the programming environment such as tracing and tutorial packages then this will allow users to concentrate on developing the basics of programming and techniques such as recursion and iteration rather than on the commands necessary to carry out their aims. A consistent and simple user environment working in terms of the user's code would act as a good base upon which to build conceptual models of the programming language and techniques therein.

A possible answer to the problem of learning dynamic programming concepts is to produce an environment which contains tools that display these concepts in such a way that their dynamic nature is clear, and the representation is understood by a novice with only a few hours experience of the programming language in question. With a display of the traced code shown in terms of the code written in the editor the user should have a more effective means of description of the programming techniques initially taught as an abstract theory. This will add some positive feedback to the programming environment, aiding both learning and debugging. What is needed is a way of representing dynamic concepts in a manner that is easy to understand, and at a level that requires little or no mental translation by the user. This will allow concepts such as variable binding, flow of control, and techniques like recursion and iteration to be demonstrated dynamically by stepping through example programs.

This thesis investigates a representation of program execution in an explicit dynamic form, with the intention that the user will be able to assimilate knowledge much more easily than from traditional tracers and single steppers. The proposed representation of program execution potentialy offers a triple aid to novice programmers :-

1) Students should be able to learn through visual examples what is actually happening in the program. These visual examples are in effect a concrete embodiment of the workings of a program (c.f. Mayer, 1979; Lieberman, 1982).

2) Students should be able easily to debug their mistakes by understanding them in terms of the evaluation of the code.

3) Students should be able to develop their programs more quickly due to the ease of monitoring the surface evaluation of the program.

We intend to test the first assertion, which claims that the proposed animated representation of a trace will improve the communication of run-time information, allowing novice programmers to build an accurate conceptual model of the workings of a language. It is felt that this is the most important of the three assertions because the other two are dependent upon the effectiveness of the first. In other words unless this approach to tracing is successful in communicating run-time information to the novice, then it is unlikely that it will aid novices in the debugging and development of progams. On the other hand if this method of tracing proves successful then the improved communication of information provides the basis for both the second and third assertions. In order for novices to debug and develop programs they must understand both the current actions of that program, and the actions of any changes they propose to make. It is claimed that this understanding will be increased by an animated trace package.

The representation of program execution described here is in the form of a direct copy of the user's 'edit-time' code so that the novice can see the correspondence of the 'trace-time' code to the 'edit-time' code. In this research 'edit-time' code refers to the code that the user has written in the editor or at the top-level, whilst 'trace-time' code refers to the simulated evaluation of that 'edit-time' code.

The area this thesis focuses on is plugging the conceptual gap, which is left by the traditional methods of teaching programming languages. The problem arises because students are normally taught the concepts of a programming language and programming techniques in an abstract manner. This abstract manner usualy takes the

form of an algorithm, in terms of a flow diagram or even plain English. The difficulty that the student faces is that of translating this abstract information, which describes a hypothetical program or programming technique, into the 'nuts and bolts' of a particular syntax and set of commands.

The aim is to solve this problem by bringing code and technique together to show the student, working concrete examples of both programming techniques and the way particular programming languages work. This will be carried out with a dynamic/animated representation of the execution of the code, in terms of the code that the user has written in the editor. This animated representation or 'visual model' of the language in action, which is henceforth called 'APT' (Animated Program Tracer) will enable the user to view the action of a program in a way that traditional tracers and single steppers do not allow, and will bridge the conceptual gap mentioned above. The proposed approach will allow the execution of a program to easily be followed by the user. The 'visual model' of program execution provides a basis for the novice to associate the 'nuts and bolts' of a program to the algorithm or programming technique that the novice is attempting to learn/program

In detail APT shows how variables get their value; how parameters are passed; the flow of control; the relationship between the edited code and the traced code. It is based on the model of the language that is presented in the majority of programming manuals i.e. the surface evaluation of the program code.

## 1.3 Overview

This thesis is concerned with the principled design of a computational environment which depicts an animated view of program execution for novice programmers. The approach taken is that being formed by the new discipline of 'Human-Computer Interaction', which is exemplified by the journal of the same name. This interdisciplinary approach has as yet not developed its own methodology

but borrows freely from the fields of 'Artificial Intelligence', 'Cognitive Psychlogy', and 'Computer Science' in an attempt to improve the state of knowledge about computer interfaces based on the perceived problems encountered by computer users. The aim of this thesis then, is to develop a more systematic, if not yet scientific, basis for the design of animated tracing tools.

The text is divided into eight chapters. Chapter 1 is this chapter, the introduction, the other seven are as follows:

Chapter 2 discusses work related to this thesis from three different areas of research. The first section looks at studies of the behaviour of novice programmers, and discusses the problems novices encounter when they learn a new programming language. The second section discusses programming environment design issues that have been built up over the years and attempts to relate these principles to the domain of animated tracing tools. The last section of chapter 2 looks at programming environments that have attempted to solve some of the problems presented in section 1 with a visual presentation of program execution.

Chapter 3 introduces the design principles for APT, drawn from a combination of the literature described in chapter two and some new ideas aimed at reducing/solving the problems encountered by novice programmers discussed in Chapter 2.

Chapter 4 presents prototypes of APT constructed for three languages: Prolog, Lisp and 6502 Assembler. This shows the generality of the design principles detailed in chapter 3 to any programming language, and tests the effectiveness of this approach in communicating information about program execution to the novice.

Chapter 5 consists of an evaluation of the prototypes described in the previous chapter. This evaluation looks at how successful each prototype is in communicating run-time information to novice users.

Chapter 6 discusses the details of a Prolog implementation of APT and presents a scenario of APT in use.

Chapter 7 comprises experiments to determine what misconceptions novices build up about the Prolog interpreter, and how APT affects the ability of novices to understand the actions of Prolog programs at run-time. This second experiment tests the validity of the first assertion made above concerning an animated approach to viewing program execution.

Chapter 8 offers a critical appraisal of APT, and discusses possible future directions for research in this area. Figure 1.1 presents a summary of the experimental studies carried out in this thesis along with page numbers showing where they may be found.

| Name of Study | Experimental Technique | Page in thesis |
|---|---|---|
| Prolog prototype | Canned tracer/timing and verbal protocols | 5-5 |
| Lisp prototype | Canned tracer/timing and verbal protocols | 5-11 |
| Assembler prototype | Canned tracer/timing and verbal protocols | 5-18 |
| Explanations experiment | Questionnaire | 7-8 |
| Misconceptions experiment | Questionnaire | 7-12 |
| APT experiment | Questionnaire | 7-41 |

Figure 1.1 Summary of Experimental Studies contained in the thesis

# CHAPTER 2
# RELATED RESEARCH

## 2.1 Introduction

This chapter discusses work related to building animated tracing tools from three different research areas.

The first section is concerned with the behaviour of novice programmers when either learning a programming language for the first time, or learning a new language. This section determines what the question is that animated tracing tools must answer, and shows why they are necessary, by specifying the novices' difficulties.

The second section looks at the relevant literature in order to pick out well known design principles from other domains which may be relevant to dynamic tracing tools. The intention is to combine these principles with those found in the third section to form a comprehensive set of principles for the design of dynamic tracing tools.

The third and final section describes the attempts that have already been made at producing some kind of animated tracing tool, whether for novices, for experts or for both. The features of these systems which seem most appropriate in communicating dynamic concepts are extracted in order to gather together a set of design principles for building dynamic tracing tools.

## 2.2 The Programming Behaviour of Novices

This section will take both a high and low-level view of what happens when novices learn a programming language. The first part discusses the notion of a 'conceptual model of program execution' which novices build when programming. The second part looks at research into the specific problems novices have when learning a programming language.

### *2.2.1 Understanding Programs*

Lukey (1980) in developing a system to understand and debug simple Pascal programs defines what it is to understand a program:

> Before designing a mechanism to understand programs, one must first decide what it is that constitutes an understanding of a program. This is a very difficult problem. The theory views the understanding of a program as the construction of descriptions of the program. These descriptions should indicate what the program does and how the program does it.[p189]

Although this description of program understanding is aimed at machine understanding, I believe it to be equally relevant to human understanding.

Studies on the knowledge held by expert and novice programmers (Soloway, Ehrlich, Bonar and Greenspan, 1982; Soloway and Ehrlich, 1984) shed some light on how this program understanding might take place, and show the gap that exists between experts understanding of programs and that of novices.

Soloway et al suggest that experts posses two types programming knowledge, (i) 'Programming Plans', and (ii) 'Rules of Programming Discourse', while novice programmers have no such high level knowledge. The first of these consists of 'program fragments that represent stereotypic action sequences in programming', such as a running counter plan. This suggests that experts hold a library of program fragments which they draw upon when developing a program. The second type of knowledge concerns 'rules that specify the conventions in programming', which 'set

up expectations in the minds of the programmers about what should be in the program'. An example of a discourse rule is to use a variable name that corresponds to its function. The authors suggest that these rules are analogous to rules of discourse used in conversation.

It is these 'Programming Plans' and discourse rules that experts possess that enable them to understand programs, and that novice programmers need to learn in order to increase their understanding of programming.

Lukey's description of program understanding essentially suggests that a novice must build up a description of the execution of the program in terms of what the program does, and of how it does it. Soloway et al suggest that this description of program execution is built up of program fragments and rules of discourse.

### *2.2.2 The Model of the Language Presented to the User*

The ideas mentioned above resemble closely the idea of a 'mental model' which several researchers believe novices build when learning to program.

du Boulay, O'Shea and Monk (1981) present the concept of the 'notional machine', which they describe as, 'the idealized model of the computer implied by the constructs of the programming language'. Norman's (1982) 'system image' presents the same idea as the 'notional machine' (from here on the term 'notional machine' should be taken to mean 'notional machine' and 'system image'). This 'notional machine' consists of every aspect of the computing interface the user comes into contact with that relates information concerning the action of the programming language. This includes the user guide, the programming language, and the programming environment on the computer, but not the hardware of the computer.

In other words the 'notional machine' consists of a model of the execution of a programming language, from which the user attempts to determine how the language

works and thus build their mental model of execution for the programming language they are learning.

The descriptive level at which the 'notional machine' is presented to the user is not specified by Norman or du Boulay et al. The problem arises that the description could be at several different levels each of which would constitute a 'notional machine'. For example a programming language may be described in terms of (i) the language it is implemented in; (ii) the programming language itself; (iii) an analogy. The first two points are discussed in more detail in section 3.2.4, while the last point is covered in section 2.2.6. Each of these descriptions will tell a different story of program execution, so care must be taken when interpreting the phrase 'notional machine'. For the purposes of this thesis 'notional machine' should be interpreted as a description of how a programming language executes at the same level at which the language is presented in programming texts.

### *2.2.3 The Effect of Presenting the User with a 'Notional Machine'*

There is evidence to support the usefulness of presenting the novice with a 'notional machine' from which s/he may build up a mental model of program execution. Mayer (1975,1981) conducted experiments which showed that when novices were given a description of their first programming language's 'notional machine', they learned programming more effectively than those who had no such aid. The description given to subjects consisted of a four foot diagram which made visible the basic operations of the computer to the subject, plus a one page description. This model presented a concrete analogy for the four major functional units of the computer,

> (1) Input is represented as a ticket window at which data are lined up waiting to be processed and placed in the finished pile after being processed; (2) output is represented as a message note pad with one message written per line; (3) memory is represented as an erasable scoreboard in which there is natural destructive read-in and non-destructive read-out; (4) executive control is represented as a recipe or

> shopping list with a pointer arrow to indicate the line being executed.[Mayer 1981, p128]

This shows that novices can learn the concepts inherent in a programming language more easily/quickly if they are presented with a 'notional machine' (a description of how the language works) than if they were not. The model of the computer presented by Mayer is a static picture of the computer, attempting to portray the dynamic actions at run time. Although this model improves the performance of novice programmers, it does not tell us which is the best way to portray the 'notional machine' for novices to assimilate an understanding of the concepts of computer programming

### *2.2.4 Concrete Examples*

Jones (1984a; 1984b) in studying how novices build up their mental representation of program execution, states that

> A particular problem for novices learning their first programming language, therefore, is the lack of an appropriate cognitive framework which will serve to relate new information to existing knowledge.[Jones 1984a, p777]

She shows how ideas such as the 'notional machine' attempt to provide such a framework for novices to build on. Jones, through experiments with novices learning SOLO (a language specifically designed for them) concludes that novices often build inaccurate models of program execution even when presented with the 'notional machine'. She states,

> For example, subjects spontaneously use their own metaphors to relate new information to existing knowledge, and as we have seen, this can lead to inappropriate expectations.[Jones 1984a, p782]

Jones reports that novices have difficulty coping with the concepts of flow of control, and procedure calling, especially the actions of the computer in executing control statements and procedure calls. This affects both their ability to write programs and their understanding of programs when they are run.

One pointer which Jones' data gives towards improving novices' performance in learning to program, is that '*although novices do not find it easy to abstract plans from examples that are given, programming success seems to be dependent upon doing so*'. It seems therefore, that it is important to present novices with concrete examples which embody these abstract plans, so that they will pick up the abstract plans in terms of a concrete program which they will find easier to understand. The 'notional machine' should be capable of presenting the abstract plans (algorithms) mentioned above in a concrete fashion. In order for the 'notional machine' to present an algorithm successfully it must present the novice with a dynamic view of the algorithm in action. Anything less will cause misconceptions in the user's conceptual model due to the lack of detail, and mismatch between the dynamic concept and the static presentation.

### *2.2.5 Novice vs Expert Debugging Strategies*

Research looking at how novice and skilled programmers differ in their ability to debug programs (Gugerty and Olson, 1986) shows that there are differences in the strategies that each group use. Both groups studied the buggy programs intensively before any action was taken, the novices doing the same things as the skilled programmers. One difference between the two groups was that the novices took longer in studying the programs than did the skilled group. Another difference between the groups occured in the way they attempted to solve the bugs in the programs. The novices seldom found the bug in their first attempt at changing the program, and often added aditional bugs. The experts on the other hand usually found the bug at the first attempt, and almost never added bugs to the program.

According to Gugerty and Olson, the reason the novices had such difficulty in correcting the buggy programs was that the novices generated inferior hypotheses about the behaviour of the program, and it was the inferior hypothesis that led the novices to make unnecessary changes to the program rather than to correct the original

bug. The reason given for the experts' superiority in debugging programs was the "ease with which they understood what the program does and what it is supposed to do". Conversely novices did not possess sufficient programming knowledge to generate a good hypothesis for the program's behaviour. In other words the novice did not comprehend the program whilst the expert did.

This evidence supports the research described earlier which suggests that the main obstacle novices find when learning a programming language is the assimilation of a model of the 'notional machine'. The lack of this model of how the system works is made apparent in the experimental findings of Gugerty and Olson where the lack of comprehension of program behaviour causes novices problems in debugging programs.

### *2.2.6 The Effect of Analogies upon the Users Conceptual Model*

The above research has shown the importance for the novice of building a conceptual model of the programming language, which will allow him/her to make predictions about the actions of that language. It has also shown that providing the novice with a 'notional machine' upon which to base this conceptual model facilitates the learning of that language. What we must do now is to determine how this 'notional machine' should be presented to users so that they can build their conceptual model more efficiently and thus have fewer misconceptions creep in to the model.

It is generally thought that using metaphors and analogies is a useful way of introducing novices to programming (Rumelhart and Norman, 1981; Carroll and Thomas, 1980), and with the advent of the desktop metaphor increasingly used as the user-interface in today's computers, this approach seems very convincing. However there is research which shows the deficiencies of using metaphors and analogies in teaching complex concepts.

The argument in favour of analogies is descibed thus by Halasz and Moran (1982)

> Given the analogy, the new user can draw upon his knowledge about the familiar situation in order to reason about the workings of the mysterious new computer system. For example, if the new user wants to understand about how the computer file system works, he need only think about how an office filing cabinet works and then carry over this same way of thinking to the computer file system [p383]

But the problem with analogies is that because they are analogies the story they tell will only be accurate some of the time, and when an analogy is used to describe a complex system it will not be specific enough to guide the user to the actual operation of the system (Young, 1981). Halasz and Moran also point out that analogies are bound to contain many aspects which are irrelevant to the system being learned, and they will also only cover a small part of that system (the more complex the system the less the analogy will cover). The problem here is that the novice will not know which part of the analogy to believe and which to ignore. This in turn is bound to lead to misconceptions in the model of the machine which the novice is building. For the novice, there is no reliable one-to-one mapping from any part of the analogy to any part of the domain being learned.

Instead of using an analogical model, Halasz and Moran suggest that, 'the appropriate basis for teaching about a computer system is an abstract conceptual model' which will 'provide a specialised framework for reasoning about the system'. In other words this means directly presenting the user with the underlying conceptual structures of the system being learned, and it is these that will provide the user with an appropriate basis for reasoning about the system s/he is learning.

### *2.2.7 Problems with the Presentation of the 'Notional Machine'*

The above approach, suggested by Halasz and Moran, is similar to the idea of presenting the 'notional machine' or 'system image' to the novice (discussed in section 2.2.2) in an attempt to help them build a conceptual model of the programming language. However the 'notional machine' still poses a problem to

novices. In its present form it provides the novice with a model of the programming language drawn from the programming environment, its commands and its manuals. It may be the case that, because the examples which are given are presented in a static printed form, that novices cannot see the important features that these examples are intending to exemplify. Trying to associate these static examples with actual running programs is not easy. If the examples were presented in a fashion more in accord with program execution the novices might well glean more information from them, so picking up the more complex programming concepts.

Although du Boulay et al say that wherever possible methods should be provided for the learner to see the workings of the 'notional machine' in action, the methods they discuss do not go far enough. Novices need to be presented with a detailed description of program execution, presented at the correct level (see chapter 3), so that they may associate the 'notional machine' with actual running programs. As Halasz and Moran (1982) say, if the user is provided with a framework around which to build a mental model of program execution, and this framework is presented in terms of the underlying conceptual structures of the system being learned, then the user will have an appropriate basis for reasoning about the system s/he is learning.

### *2.2.8 Problems Encountered by Novice Programmers*

There is a growing amount of research concerned with the specific problems novices have when learning a programming language. Over the years many ideas have been adopted to improve the performance of the programmer; however most of these aids have been adopted without knowledge of whether or not they solve any of the problems that programmers face. Sheil (1981) in a review of the empirical research studying such user aids as prettyprinting, flowcharting, variable naming, specification of data types, and commenting of code, concluded that there is no evidence to show that any of these aids actually improve the performance of

programmers. Where differences have been found between experimental conditions, Sheil comments that,

> many of these effects tend to disappear with practice or experience. This raises some doubt as to whether these results reflect stable differences between notations or merely learning effects and other transients that would not be significant factors in actual programming performance.

So what are the problems which novice programmers have and which these aids have attempted to solve?

Lewis (1980) in a long term study of novice programmers (1144 hours of computer time) using SOLO, the same language used by Jones, catalogued the errors that occured. Ignoring language specific errors the main categories of errors are listed below:

- Flow of control
  Flow of control statement used as an argument. Inappropriate use of flow of control statement. Missing control flow statement.
- Side effect
  Attempting to assert into the database facts that already exist. Attempting to delete facts that do not exist.
- Recursion
  Infinite loops.
- Procedure call
  Too many/few arguments.
- Variables
  Non-unique variable names used as global variables.
- Modes
  User assumes that s/he is at the top-level when in the editor, and vice versa.

Of the errors mentioned above most are dynamic in nature, in the sense that they are due to misconceptions of dynamic concepts. These errors only become apparent to novices at run time, as they lie hidden in the static environment of the editor. Likewise the misconceptions novices have of dynamic concepts lie hidden with the static teaching methods used in text books, on blackboards, and in most on-line

tutorial packages that are available. One way to remove these misconceptions or even prevent them arising is to provide the novice with a dynamic environment in which to present the type of dynamic concepts which at present cause them problems.

Many other studies have been conducted to investigate the type of problems that novice programmers meet in their first few weeks programming. There are several studies on the problems novices have with control flow statements (Miller, 1974; Mayer, 1976; Sime, Arblaster and Green, 1970). Domingue (1985) reports similar errors to those mentioned above for novices learning Lisp. Others have reported the great problem that recursion causes novices (Kahney, 1982; Anderson, Pirolli and Farrell, 1984; Pirolli and Anderson, 1985). Kahney found that although many novices had some model of recursion, it was an incorrect model. This model was sometimes in fact based on a correct model of recursion, and other times based on something more akin to iteration. He also found that some novices had no model of recursion at all.

Soloway, Ehrlich, Bonar and Greenspan (1982) have shown that novice programming errors are not only due to 'difficulty with syntax and semantics' of the language, but also to problems in 'determining which constructs to use and how to coordinate them into a unified whole'. This shows that there are two separate problems facing novice programmers. The low level problem with the syntax and semantics of the programming language, and the higher level problem of constructing meaningful units, or program fragments, when programming. These problems correspond to the notions of discourse rules, and programming plans discussed in section 2.2.1.

### *2.2.9 Summary and Discussion*

The studies discussed in the above section provide valuable information in the task of determining what the novice needs to aid his/her path through the jungle of learning a programming language.

Lukey (1978) believes that the ability to understand programs depends on the construction of descriptions indicating what the program does and how it does it. Soloway et al (1982; 1984) suggest that this description is built up from program fragments, which represent plans carrying out stereotypical actions, and rules of discourse for the programming language. This resembles closely the idea of a 'mental model' which is built up by the novice and contains information concerning the way the programming language works. The 'mental model' built by the user is extracted from the information presented by the 'notional machine' or 'system image', which describes the actions of the programming language being learnt.

Mayer (1975) has shown that the presentation of a 'notional machine' improves the performance of novices learning a programming language, allowing them to learn the concepts inherent in a programming language more easily/quickly. The suggested reason for this is that when novices learn a language they build a mental model which allows them to make predictions about the actions of that language. The 'notional machine' provides a strong basis on which to build this mental model, and allows the novice to make rapid progress as a programmer.

It is clear then that in order to aid a novice in learning a programming language it is important to present him/her with a 'notional machine'. This information will provide the basis on which the novice programmer may build his/her mental model through which s/he may understand the workings of the language.

However, Jones (1984a; 1984b) states that even when novices are presented with a 'notional machine' they often build inaccurate models of program execution. Jones has shown that novices specifically have difficulty with dynamic programming concepts such as procedure calling and flow of control. Other experimental studies on novice programming behaviour (see section 2.2.8) support this and show that the subject matter of the problems novices encounter is dynamic in nature. Examples of this include variable binding, flow of control, recursion and side effect errors.

Jones goes on to point out that even though novices find it difficult to abstract plans from the examples they are given, that their success as a programmer is dependent upon doing so. This ties in with the importance placed upon 'Programming Plans' in understanding programs by Soloway et al, and suggests that in teaching novices how to program it is essential that they be presented with concrete examples which are related to particular abstract 'Programming plans'.

We can now see that the 'notional machine' that we give the novice must take into account the difficulty they have with dynamic programming concepts, which relate to Soloway's discourse rules, and also the problem they find in relating the abstract plans (algorithms) to concrete examples, which relates to Soloway's 'Programming Plans'.

Gugerty and Olson (1986) claim that the difference in the ability of experts and novices to debug programs is due to the hypotheses produced by each group concerning the behaviour of the program in question. The hypotheses produced by the novices are inferior to those of the expert because they do not possess sufficient programming knowledge. While the experts understand what the program does and what it is supposed to do.

If novices do not understand what programs do it is because they do not possess the discourse rules and 'Programming Plans' suggested by Soloway et al. The reason that novices lack this knowledge, as Jones points out, is because of the difficulty that they face in relating abstract plans to concrete examples. In order to correct this problem novices need to be provided with a mechanism by which they can see what concrete programs are doing and have them related to abstract plans.

Halasz and Moran (1982) show that the traditional method of using analogies and metaphors to teach programming concepts can cause novices problems instead of aiding learning. This is because analogies can at best only explain a part of a system, and will contain many features which are not only irrelevant to the system being

learned, but inaccurate as well. They suggest that the user should be presented with the underlying conceptual structures of the system rather than an analogy.

Thus, the method of presenting the 'notional machine' to the novice user should not be an analogy of the programming language, but rather show the underlying conceptual structures of the system, which will give an accurate story of how the system works.

Drawing all these threads together provides a clear vision of how to approach the problem novices face when learning a programming language. The novice needs to be provided with a basis for understanding how programs work, both at the level of syntax and semantics and how programming constructs are used in combination to build programs.

The traditional 'notional machine' presents the novice with a static view of the programming language because it is based on text books or disparate tracing snapshots. In order to give the novice the information necessary to understand the dynamic concepts, it is necessary to show the 'notional machine' in action. In other words, instead of presenting the novice with a static 'notional machine', giving them a dynamic view of program execution should communicate the type of information required to build a mental model of the workings of the language.

This dynamic 'notional machine' must not be an analogy of the system in question, but a direct view of the way in which the program is executed. This will provide an accurate story with which the novice can build his/her mental model. Also the way in which this view is presented must enable the user to see concrete examples of 'programming plans' so that s/he can abstract out the way constructs are combined to produce a program.

An animated tracing tool that showed details of the execution of programs would provide the user with a dynamic 'notional machine'. This would directly show what a program does, and how it does it. In other words novices would have a basis for

understanding the syntax and semantics of the programming language. This type of tool would also provide a means of presenting concrete examples of 'programming plans' which should facilitate the novice programmer in determining how constructs are combined to form programs that embody abstract algorithms.

From this improved 'notional machine' the user should be able to build an accurate mental model of how a programming language works, not only in terms of the syntax and semantics, but also of 'programming plans'.

## 2.3 Design Principles for Programming Environments

This section presents the research on system design principles. The material on this subject is somewhat sparse as Norman (1983b) states,

> Today, proper tools for design do not exist. Thus, the designer who wishes to minimize error has no standard reference to turn to for advice.[p254]

The material concerned with system design principles that does exist is aimed at a fairly general level, and is rather abstract in nature. Each of the following sections discusses particular design guidelines, and how these principles can be related to the domain of animated tracing tools.

### *2.3.1 Glass Box (du Boulay, O'Shea and Monk, 1981)*

'The black box inside the glass box: presenting computing concepts to novices' is concerned with easing the burden on novices learning programming languages. The main points are that the 'notional machine', described in section 2.2.2, should be kept *simple*, *consistent* and made *transparent*..

Simplicity means that the way in which the environment is presented to the user should be limited in the number of transactions needed to explain the events that occur in the environment. Ideally the system should allow novices to carry out actions by writing simple programs, and the language should have a syntax which is uniform and has no special cases. A system with a small number of commands, and transactions necessary to explain the actions of those commands, should be easy for the novice to learn due to the small amount of information.

Consistency means that every way the system is presented to the user should be consistent with each other. Also each command should be consistent in its action throughout the programming environment. A consistent system means that the novice

can use information learned in one place in order to predict what will happen anywhere else in the system.

Transparency is concerned with displaying the 'notional machine' to the user so that s/he can see in detail what happens when programs are executed, enabling the user to understand the consequences of the commands in the program. Because the novice can easily see what happens when programs are executed s/he should find that this view helps him/her to understand and predict the actions of other programs, and ultimately help in building a working model of the system.

Although these principles aim to facilitate novices learning a computer system, it might not always be possible to combine these principles in the systems design. For example if an attempt is made to present a very complex system, such as UNIX™ or ADA, in a 'consistent' way then it might not be possible to also do it in a 'simple' manner. This is because the system encompasses so many concepts that the 'notional machine' that explains them will likely be large and complex. In the same way if such complex systems are made transparent to the user the resulting view may not be simple. However for those programming languages that have a concise conceptual basis it should be possible to present a notional machine that is 'simple', 'consistent' and 'transparent'. Where this is not the case then these principles will have to be traded-off against each other with priority depending on the end user of the system.

The above three principles should form the basis of any programming tool or environment for novice programmers because if these guidelines are adhered to then the system being designed should become so natural to use that users will not have to think in terms of how to use the system, but rather will be left free to think about the domain they are working in.

In terms of animated tracing tools these principles can be thought of on the following way.

Simplicity: The commands for an animated stepper will include those to carry on stepping, and to stop the process. In addition to this there may be the command to retrace a particular part of the program. There is no need for any more than these three commands

Consistency: The manner in which each of the languages features are displayed must be consistent, so that the novice can build an accurate model of how that feature works. Likewise the manner in which the tracing system is presented must be consistent, so that the novice can forget about the system and concentrate upon the animated display of program execution.

Transparency: This is perhaps the most important of the three principles for the presentation of animated tracing tools, because the aim of this sort of system is to make visible to the novice the actions of a programming language at all times, and in a manner which clearly shows all aspects of the 'notional machine'. In order to make all aspects of the language visible it is necessary to give a dynamic view of program execution, because many of the things that happen at this time are dynamic and a static view will not provide the transparency required. Also the animated tracing system should be transparent so that in a similar way to the principle of consistency the novice can ignore the command structure of the system and concentrate on the information the system presents.

Another point brought up by du Boulay et al is that of integrating the utilities of a programming environment so that the command structure is the same for each utility. This blurs the distinctions among utilities, but reduces the number of commands the novice has to learn in order to use a programming environment. As the novice has to use an editor to write his/her initial programs this suggests that the editor and the tracing system should be integrated. This should reduce the cognitive burden placed on the novice.

### *2.3.2 User-Centred Systems Design Principles (Norman, 1983a)*

Norman gives four strategies for producing design principles, and five slogans to guide development work. The strategies and slogans are presented below followed by a description of how they relate to the domain of animated tracing tools.

1) Be aware that the end user has special needs.

2) Provide methods and guidelines for design, preferably quantitative.

3) Provide software tools for interface design, for example to help monitor the consistency of the system.

4) Separate the interface design from other programming tasks. The interface should be a separate module so independent work can be carried out on it.

The needs of the end user of an animated tracing tool are for a clear view of the dynamic concepts embodied in program execution so that s/he can build a model of these concepts. The design principles for such a system will be detailed in chapter 3, and are being drawn from the research described in this chapter. The last two strategies are specific to the implementation of systems and are not relevant at this stage.

The slogans used to guide every day work are:-

- "There are no simple answers, only trade-offs."
  Each application of a principle has strength and weakness, and effects some other principle in some way.

Each of the design principles for building a dynamic tracer must be looked at in combination with all the other principles to determine how they trade-off against one another. This will allow the designer to build a tracer most suited to the user.

- "There are no errors: all operations are iterations toward a goal."
  An error by a user is considered as part of his/her attempt at a goal. The task of the designer is to aid the user to attain that goal.

An animated tracer should not give errors other than an error due to the program being traced. These errors should be presented to the user in the context of the program being traced so that the user can recognise the point where the error occured.

The error message given by the system should be as specific to the user's code as possible.

- "Low level protocols are critical."
  This refers to the actual operations performed by the user. If these protocols (actions) can be made consistent the ease of use of the system will increase.

The amount of information/knowledge needed by a user of a tracing system should be very small, and the functions of the commands obvious and simple. This will enable the novice to use such a system straight away rather than wait until s/he has learnt the fundamentals of a more complex system.

- "Activities are structured."
  The grouping of user goals should be made explicit to the user and the system. This may allow the system to constrain the interpretation of user inputs, using context, and be able to point the user in the right direction if help is needed.

The oustanding and previous goals in the execution of a program should be made explicit to the user at all times, and in the context of the program. In other words all tracing information should be related to the user's program as the program is traced.

- "Information retrieval dominates activity."
  Using a computer system consists of forming an intention, choosing an action, specifying the action, and evaluating the outcome. This is dependent upon the user's memory. The memory of a user should be aided by the system.

The important point here concerns the communication of information to the user. Novices should never be in a position where they are lost, or confused because they cannot remember some piece of information. The system should provide enough information for the user to determine what is happening and why. In terms of a tracing system this means that the system needs to provide both a history of the trace so the user can retrace a confusing piece of program code, and a continual brief textual commentary on what the tracer is showing.

### *2.3.3 Studies of Programming Slips (Norman, 1983b)*

Norman has studied the errors people make when using computer systems, in order to develop design principles which will minimize the occurrence of these errors. The class of errors studied, called 'slips', are defined to be situations in which the user's intention is correct, but the results do not conform to the original intention. The errors Norman found and the proposed solutions are detailed below.

*Mode errors* happen when the user carries out an operation appropriate for one mode when in another mode. For example such errors occur in certain computer text editors where users try to issue commands while still in 'text mode'.

It is stated that modes are necessary in all but simple systems, and that mode errors result from inadequate feedback as to the state of the system. The three given solutions to mode error are:-

a) Do not have modes.

b) Make sure that modes are clearly marked.

c) Make the commands required by different modes different, so that a command in the wrong mode will not lead to difficulty.

Modes are analogous to different language features such as 'cond' and 'do' in LISP, or 'backtracking', 'database search' and 'cut' in Prolog. Modes and language features are analogous in the sense that because each feature works differently the user needs to think about different things when working with features, just as s/he does when working in different modes. As it is not possible to remove these features it is essential that they are clearly marked when they occur. This can be done with a textual commentary, and with some form of highlighting of the program code in order to associate the text to the program.

*Description errors* occur when there is insufficient specification of an action and the ambiguity that is caused leads to error. This is usually due to different actions having similar descriptions which makes possible the ambiguity. For example the use

of keys for editing commands, where 'd', 'D' and 'control d' have different actions. The three principles to get around this problem are stated below along with their computer versions:

> 1. Arrange instruments and controls in functional patterns, perhaps in the form of a flow chart of the system.
>
> 2. Use 'shape coding' to make the controls and instruments look and feel different from one another.
>
> 3. Make it difficult to do actions that can lead to operations that have serious implications and that are not reversible.
>
> 1C. Screen displays and menu systems should be organized functionally.
>
> 2C. Design the command language or menu display headings to be distinct from one another so as not to be easily confused, either in perception or in the action required.
>
> 3C. Make it difficult to do actions that can lead to operations with serious implications and that are not reversible.[p256]

The dynamic descriptions of the different programming language's features should be distinct, so that the user can see each feature and recognise what part it has to play at run time. As with the previous section this can be done with the textual commentary and highlighting of both the edit-time and trace-time code.

*Lack of consistency errors* occur due to a lack of consistency of the command structure. The reason these errors happen is that when users lack knowledge about a particular operation they will try to derive it by analogy with other similar operations. This procedure fails when the system is inconsistent and operations do not map onto each other. The only way around this problem is to take care when designing a system that consistency is kept in mind at all times.

Errors can arise simply from command sequences overlapping, producing confusion among commands. In this situation the infrequently used command gives way to the more frequently used command causing an error. This can be avoided by preventing overlapping sequences, or trying to second guess the user's intention and trapping the error.

If the story that the animated tracer tells is consistent with the 'notional machine', and based on the action of the interpreter/compiler, then that story should be consistent with what happens to a program at run time. This means that the designer has to decide the level of detail of the interpreter/compiler to present in the trace (see chapter 3).

*Activation errors* happen when the user fails to carry out an operation, due to a distraction or an overlapping window, and then tries to do something else. A memory aid is necessary here to remind the user of ongoing operations, and unfinished command sequences should be repeatedly displayed.

The conclusions of this study on 'slip' errors are summed up in four principles.

> Feedback: The state of the system should be clearly available to the user, ideally in a form that is unambiguous and that makes the set of options readily available so as to avoid mode errors.
>
> Similarity of response sequences: Different classes of actions should have quite dissimilar command sequences (or menu patterns) so as to avoid overlapping and description errors.
>
> Actions should be reversible: Where actions are irreversible and of high consequence, they should be made difficult to carry out, preventing unintentional actions.
>
> Consistency of the system: The system should be consistent in structure and design to minimize memory problems, and mapping errors when retrieving operations.

To give the novice feedback about the actions of the language interpreter/compiler on a program, three things are necessary. Firstly all the outstanding, and previous goals should be shown in the trace-time code so that the user knows where s/he is in the trace. Also a history of the trace should be kept so that the user can retrace a previous piece of code without having to invoke the trace all over again. Thirdly, a commentary on the actions of the interpreter/compiler should be present at all times providing the user with a navigation aid during program execution.

### *2.3.4 GOMS Model: Goals, Operators, Methods, Selection (Card, Moran and Newell, 1983)*

Card, Moran and Newell have carried out various performance tests of systems and present ten principles of design, these are detailed below with my comments in italics:-

1. 'Early in the system design process, consider the psychology of the end user and the design of the user interface.'

2. 'Specify the performance requirements.'
   *Work out the improvements to each part of the system and consider the trade-offs.*

3. 'Specify the user population.'
   *In order to predict the performance of the system the user population must be known, to take into account their ability.*

4. 'Specify the tasks.'
   *This refers to the tasks the user will want to carry out using the system. This is needed for principle 2.*

5. 'Specify the method to do the task.'
   *This will show the interaction of methods to do the different tasks.*

6. 'Match the method analysis to the level of commitment in the design process.'
   *The level of detail in the analysis of methods must parallel the level of design being worked on at any particular time.*

7. 'To reduce the performance time of a task by an expert, eliminate operators from the method for doing the task. This can be done at any level of analysis.'
   *This means that to increase performance for experts either reduce the time taken to carry out a sequence of tasks, or reduce the number of tasks to carry out..*

8. 'Design the set of alternative methods for a task so that the rule for selecting each alternative is clear to the user and easy to apply.'
   *For example alternative methods for entering commands might be, the command name itself for novices, and key sequences for experts.*

9. 'Design a set of error recovery methods.'
   *This reduces time spent correcting errors.*

10. 'Analyze the sensitivity of performance predictions to assumptions.'
    *Assumptions about the user and system are made when doing performance analysis. It should be known how much these assumptions affect the performance figures for a system.*

The first four principles here have been discussed in the first section of this chapter. The user population are novice programmers, who are attempting to build an

accurate conceptual model of the actions of a programming language at run time. The requirement of the system is to facilitate this model building by presenting the novice with a concrete simulation of the language's interpreter/compiler working on a program, in an attempt to clarify the dynamic actions of the program that cannot be seen at edit-time. Principles 7, 8 and 10 are not relevant to this thesis, as they are concerned with the trade-off between novice and expert use and the analysis of the performance of systems. Principle 5 is concerned with how particular pieces of information are conveyed to the user, while 6 states that the level of system design i.e. design guidelines should parallel the level of interaction of the proposed system. For example an animated tracing tool will be presenting the novice with a picture of the interpreter/compiler in action, therefore any design principles should discuss the presentation of interaction at the level of the interpreter, i.e. in terms of the execution of units of program code. Principle 9 talks about error recovery from the system. In a tracing system such as this there should be no errors apart from those generated by the interpreter/compiler due to the program being traced. However such a tracing system will allow such errors to be presented to the user in the context of both the trace-time and edit-time code. This should enable the novice to recognise the meaning of the error more easily, and generate a hypothesis for error correction.

### *2.3.4 A Story of the Execution of Prolog Programs*

A set of guidelines for presenting a story of the execution of Prolog programs are currently under development at Edinburgh University (Bundy, 1983; Pain and Bundy, 1985; Bundy, Pain, Brna and Lynch, 1986). These guidelines are not principles as such, but principles for the design of animated tracing tools can be extracted from them.

Bundy (1983) states the need for a story of program execution as follows,

> When teaching a programming language it is important to give the student some model of what the computer will do with his/her programs. The student must be able to anticipate the effect of running

> his/her program, otherwise he/she will be unable to design it, debug it, modify it etc. Rather than leave the student to induce this model from examples, it is better to teach it explicitly. This will reduce the chances of the student inducing an incorrect model and will increase the chances that everybody is using the same model, and talking the same language.[p1]

The above paragraph concludes that one way to alleviate the problems novice have learning programming concepts and techniques is to present the novice with a dynamic story of program execution so s/he may build a conceptual model of the programming language, which can then provide a basis for future descisions concerning program development, debugging and the addition of more complex features and techniques.

Bundy states that a story of Prolog program execution should have the following features in order to give novices a concrete model of the workings of Prolog. While these are not design principles as such they are important points to be kept in mind when designing a system to demonstrate/teach a particular programming language, or for that matter any system.

> A good Prolog story should have the following features:
>
> - It will cover all the important aspects of Prolog behaviour, so that it can be safely used to predict the behaviour of Prolog programs.
>
> - It will be simple to understand and use, even by people with no previous computing/mathematical experience.
>
> - It will illuminate the tricky aspects of Prolog behaviour, e.g. recursion, backtracking and cut.
>
> - It will be used universally by Prolog teachers, primers, trace messages, error messages, etc.[p1]

The features detailed above are general in nature and as such are relevant to most programming languages. At a more detailed level Pain and Bundy (1985) and Bundy et al (1986) present an ideal story of Prolog program execution with the following properties:

1. the flow of control through the search space would be indicated;

2. the overall search space of the call would be conveyed, in particular, the backtracking points would be indicated, and it would be obvious when ultimate success had been attained;

3. each goal would be displayed;

4. the unifiers produced by the resolution of subgoals would be displayed;

5. the remaining subgoals would be displayed;

6. the final instantiation of the original goal would be displayed;

7. different instantiations of a clause would be distinguished;

8. the effect of a cut, on the search space, would be indicated;

9. the clauses that resolve the subgoal away would be displayed;

10. the other clauses that could resolve with the selected subgoal would be displayed.[Pain and Bundy 1985, p2]

The following principles can be extracted from the above features. To provide a view of program execution the story must show the order in which programs are executed, in a way that presents the current, future and past states of the execution. The story should be detailed enough to show everything that happens during program execution. It should be simple enough to use so novices can use the model straight away as soon as they start to program. The story must cope with all the tricky language dependent features for example the 'cut' in Prolog, and should be able to be used by language primers, trace packages and error messages.

### *2.3.6 Summary*

To present a clearer picure of how the relatively general system design guidelines given above pertain to the domain of animated tracing tools, I will draw them together and present a summary below.

The end user - novice programmers

Aim - to present the user with a clear view of the dynamic actions of an interpreter/compiler at run time, so the user can rapidly build a model of these actions, from which predictions can be made about future occurrences of similar pieces of program code.

Simple - small number of commands to control the system.

Consistent - the story that the tracer tells and the way it tells it should be consistent.

Transparent - all the actions of the programming language should be made visible to the user.

Integrated system - different utilities should be integrated to reduce the amount of information a novice must know before s/he can begin to use a system.

Feedback - information as to the current and past state of the tracer should be provided to the user at all times, preventing him/her getting lost or confused.

Differentiation of descriptions - descriptions of different traced language features should be distinct in order for novices to recognise and understand their actions. Care should be taken to remove any ambiguity in these descriptions.

Errors - the tracing system itself should produce no errors. Any errors occuring from program errors should be displayed to the user in the context of edit-time and trace-time code enabling the user to understand the error in the correct context facilitating error recovery.

More specific design guidelines for animated tracing tools, and a discussion of their trade-offs will be presented in chapter 3.

## 2.4 Dynamic Tracing Tools

There are many user aids existing today to help both novice and expert programmers. These can be split into two categories:

1) user aids provided by the environment

screen editors
error messages
syntax monitoring
spelling checkers
windowed displays
pretty-printing
tracing
single-stepping
command format prompting
help systems
command abbreviation expansion
undo facility

2) user aids for testing the correctness of a program

program proving
exhaustive program testing
analysis via effects descriptions
automatic programming
heuristic analysis
algorithmic analysis

These aids are very helpful at the level of the code, allowing the user to enter syntactically correct code, and testing the application of the code to the problem. None of the aids mentioned above help the novice with higher level programming problems, such as flow of control, recursion and variable binding. Even the tracers and steppers do not help, because at this early stage in learning a programming language the novice does not understand the language well enough to be able to use these tools in an effective way.

There has been a substantial amount of research into the problems novices have in learning and understanding high level programming concepts, and into how these problems might be alleviated, and ultimately solved. This research has taken several

different directions. Firstly there is a body of work concerned with studying novice programmers' behaviour as they are learn a programing language, presented in section 2.2. Secondly there is research concerned with design issues for building new programming environments, discussed in section 2.3. Lastly there is the work carried out in implementing actual programming environments for novices. This section presents the relative success of attempts made to alleviate the problems novices encounter when learning to program by systems that in some way attempt to animate the execution of programs.

Each section consists of a description of the system followed by a critical appraisal of its success in communicating dynamic information in terms of the aims and principles mentioned in the previous two sections. At the end of the section the features of the discussed systems which are deemed to be good and bad are summarised.

### *2.4.1 Baeker (1975)*

Baeker developed animation systems for both LOGO and micro-PL1. These systems were intended for use by instructors to produce animated film clips to clarify certain program features in a classroom situation. The LOGO system was intended to portray recursion and character string manipulation, whereas the micro-PL1 system focused on iterative processes and simple symbolic computation.

The main features of the LOGO system are that it shows the order of the evaluation of a program, including the achievement of intermediate results (see figs 2.1 - 2.5).

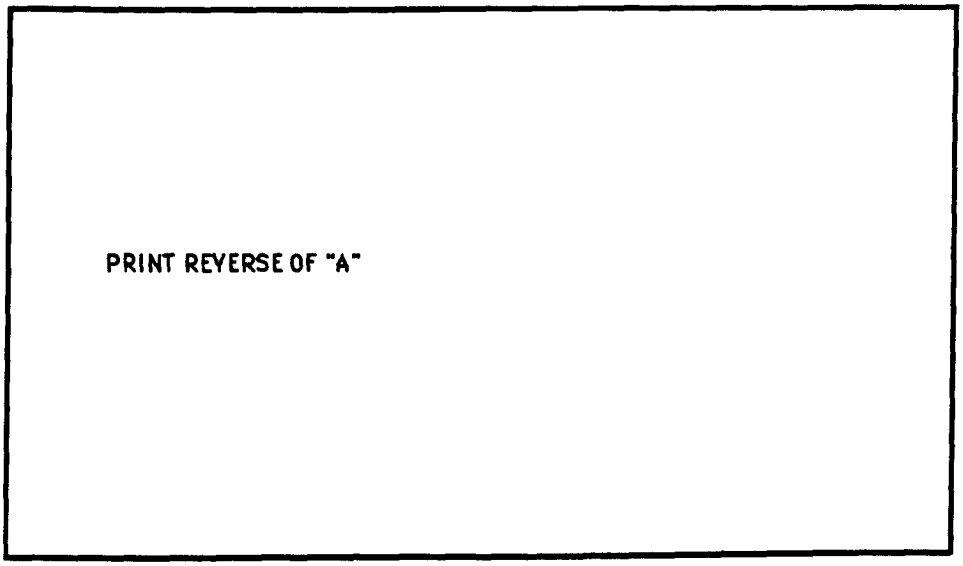



Figure 2.1 A series of screen snapshots from Baeker

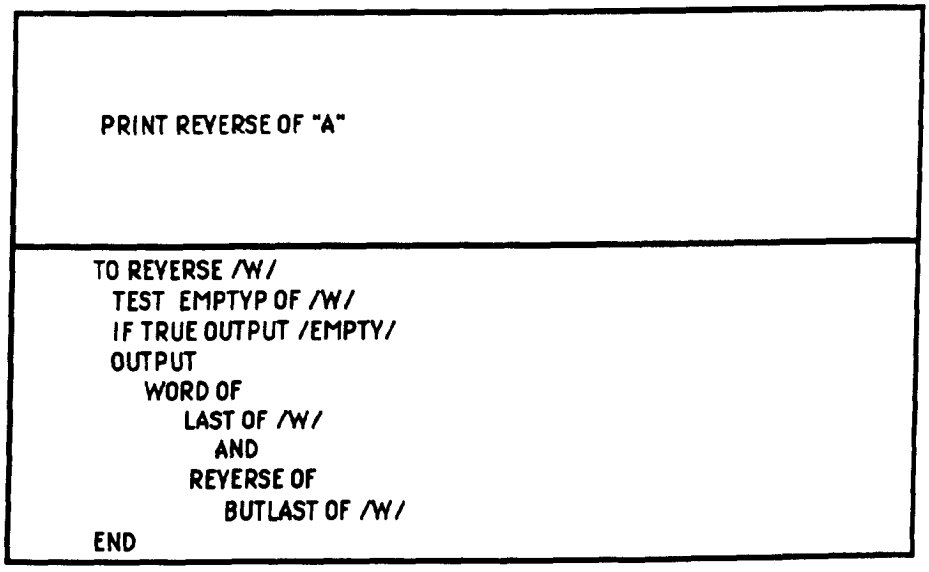



Figure 2.2

```
PRINT REVERSE OF "A"
```

```
TO REVERSE
  TEST  EMPTYP OF
  IF TRUE OUTPUT /EMPTY/
  OUTPUT
     WORD OF
        LAST OF
           AND
        REVERSE OF
           BUTLAST OF
END
```

Figure 2.3

```
PRINT REVERSE OF "A"
```

```
TO REVERSE "A"
  TEST  EMPTYP OF  "A"
  IF TRUE OUTPUT /EMPTY/
  OUTPUT
     WORD OF
        LAST OF  "A"
           AND
        REVERSE OF
           BUTLAST OF  "A"
END
```

Figure 2.4

```
TO REVERSE "A"
  TEST EMPTYP OF "A"
  IF TRUE OUTPUT /EMPTY/
  OUTPUT
     WORD OF
        LAST OF "A"
          AND
        REVERSE OF
           BUTLAST OF "A"
END
```

Figure 2.5

The display uses a representation in which the data is shown in the context of the program text. Active LOGO tokens are shown in a larger size than inactive tokens. A LOGO program would be traced in the following way:-

1) The procedure call is displayed in the centre of the screen.

2) The screen splits into two. The top half containing the procedure call and the lower half the procedure that has been called.

3) The values of the arguments in the procedure call which is displayed in the top window are bound to the variables in the procedure in the lower window. This variable binding is shown by replacing the variable name with the value it holds in the trace-time code.

4) The top window disappears, and the lower window takes over the whole screen.

5) The program is executed by replacing LOGO keywords (functions and operators) and variables with their values in a linear sequence, when they are executed. This is carried out by fading out the function and fading in the value that it returns.

The micro-PL1 system is very different from the LOGO system in that it only displays the data being manipulated by the program, although it is quite flexible as to how the data can be presented. The instructor inserts pseudo commands throughout the micro-PL1 code which determine the form of the display. For a display manipulating an array the following rules would be followed:-

1) Each array element is represented as its character string equivalent.

2) The horizontal position of each character varies linearly with its index to the array.

3) All characters have the same vertical position, horizontal and vertical size, and intensity.

4) When an array element is changed the display is updated to show this.

Both the above systems are simple to use as they are films, so the user just has to watch and learn. However this is a major drawback because they can only be used to produce animated films for classroom work, and thus students are not able to use the systems interactively at their own pace, and more importantly on programs they have written themselves.

The LOGO system shows the sequential execution of the program code in terms the user will easily recognise, i.e. the edit-time code. It is unfortunate that Baeker's system does not display a copy of the edit-time code at *all* times during the trace. If it did, this would allow the novice to associate the trace-time code with the edit-time code, or in other words the static representation of the edited code to the dynamic representation of the trace-time code. This system also has the advantage of showing how the variables become bound to their values, by replacing the variable names on-screen with the values they hold, as they become bound. This further facilitates the story of dynamic program execution.

Another general problem is that both systems address themselves to specific programming features i.e. recursion in LOGO and iteration in micro-PL1. It would be far more advantageous to have a system that could cope with all programming features of a language which would tell a complete story of the action of the interpreter on a program, otherwise the model of the language built by the user will be incomplete and lopsided.

The micro-PL1 display suffers from showing the manipulation of the data in isolation from the code, which could cloud understanding of which piece of code

produces what effect. This is not the case in the LOGO display where the edit-time code is displayed. Unfortunately this idea is not used to the full. The called procedures that are presented are changed on binding of variables and evaluation, and so it is not possible to see both the original procedure and the evaluation of that procedure at the same time. The LOGO system changes from a one window display to two windows, and then back to one window, when it moves from showing the calling environment to the tracing environment. This removes the call from the screen which means the user no longer has access to the information concerning the call, and may be distracted by the window switching. These features can be confusing to the novice programmer, and distract him/her from his/her goal of watching and learning the dynamic view of program execution.

### *2.4.2 BIP*

BIP (Barr, Beard and Atkinson, 1976) was an attempt at an automated tutoring system for teaching students to program in Basic. Although the tutoring system is not directly of interest here, BIP contains an interesting tracing tool which is described as follows:

> As each line of the program executes, the line number is displayed on his teletype or display terminal. Any variable assignments performed in that line are also indicated, as well as any input or output.
> FLOW is a more sophisticated program tracing aid designed for CRT displays. The main program is displayed on the terminal, with the text of all subroutines removed. Each time the student presses the CR key, one line of his program is executed, and its line number blinks on the screen display. When an IF or a GOTO statement is executed an arrow is drawn on the screen to indicate the transfer of control. When a subroutine is called, the main program display is replaced by the lines that make up the subroutine. Additionally, a message in the corner of the screen indicates the level of nested subroutines.[p582]

The display consists of the program being traced at the top of the screen, and any input and output is shown beneath this (see figs 2.6 - 2.9).

```
I = ?                    N = ?                    MAIN PROG
                                                  HIT <CR>
                                                  TO RUN
    10 PRINT "HOW MANY ● SIGNS DO YOU WANT
    15 INPUT N
    20 S$ = " "
    30 FOR I = 1 TO N
    40 S$ = S$ & " "
    50 PRINT S$ & "●"
    60 NEXT I
    99 END



    HOW MANY ● SIGNS DO YOU WANT
```

Figure 2.6 A series of screen snapshots from BIP

```
I = 1                    N = 2                    MAIN PROG
                                                  HIT <CR>
                                                  TO RUN
    10 PRINT "HOW MANY ● SIGNS DO YOU WANT
    15 INPUT N
    20 S$ = " "
    30 FOR I = 1 TO N
    40 S$ = S$ & " "
  _ 50 PRINT S$ & "●"
    60 NEXT I
    99 END



    HOW MANY ● SIGNS DO YOU WANT
     2
OUTPUT  ●
```

Figure 2.7

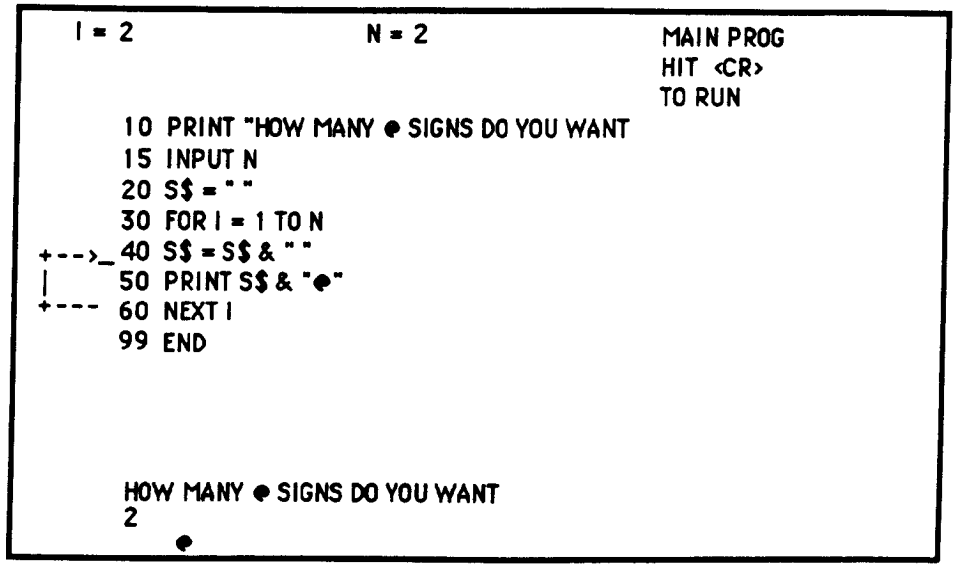



Figure 2.8

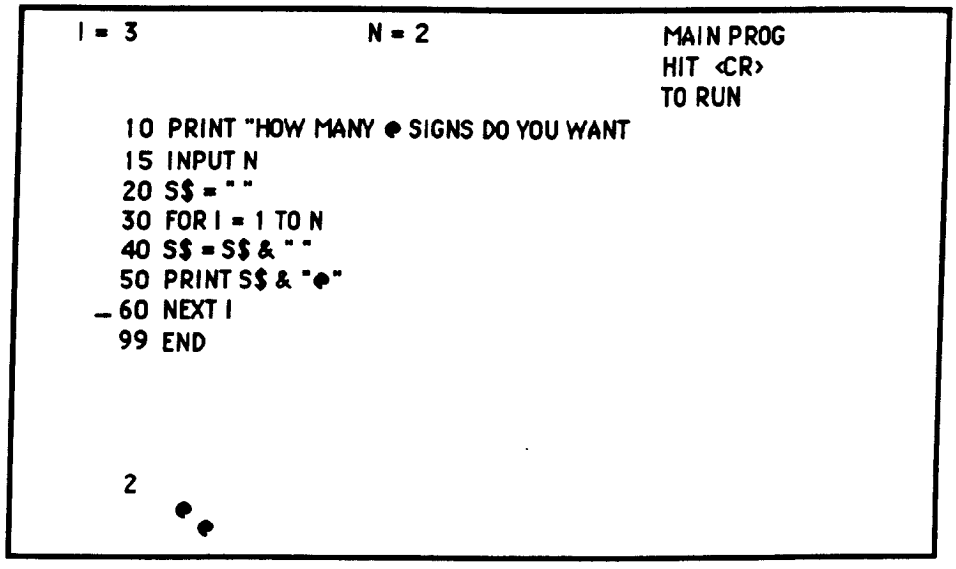



Figure 2.9

The BIP trace does not print the body of subroutines on the screen. When a "GOSUB" is executed, the screen is rewritten replacing the old code with the code referenced by the "GOSUB". These backward and forward jumps are shown with arrows connecting the relevant statements. The current line of code being executed blinks to focus the user's attention on it.

Up to six specified variables can be traced. These and their values are shown in the top right hand corner of the screen. If an array is traced then only the most recently assigned element is displayed.

If a line number is specified in the trace call then the trace animates the code up until that line number is reached, and then goes into step mode.

This tracer uses arrows to show the current focus of the stepper, in this case the control flow. But, any obtruding characters used in a display cloud the presentation of information in that display and distract the user from looking at the content of the stepped code.

BIP's story of program execution is presented as a sequential execution of the program in terms of the user's edit-time code. Up to six variable bindings are shown, albeit in a seperate part of the screen, and only then when they are specified. This will pose a problem for novices as it is unlikely that they will know which variables to trace, and may not notice what is happening to them, how variables become bound, and lastly when they become bound. During the trace the user is only shown a copy of the edit-time code. The run-time actions on the program are shown distributed around the screen. For example, the variables and their bindings are shown at the top of the screen; the movement through the program showing execution is displayed by means of arrows pointing at the copy edit-time code; input and output are shown at the bottom of the screen. To extract the run-time information form this system the user must look in three different places, and then associate the changes which occur in these places to the edit-time code. This would not happen if these run-time changes were integrated into one display showing a single story of program execution.

Because of the disjointed tracing view presented it is unlikely that the novice will be provided with a simple enough view of program execution for them to be able to extract the important features necessary to build a working model of the language. In

addition to this the novice is distracted from the important information in the display by the use of arrows as a method of pointing to the current focus of attention.

### *2.4.3 ANTICS*

The purpose of ANTICS (Dionne and Mackworth, 1978) is to introduce novice programmers to the basics of the LISP programming language. It can be used interactively by the student, or by an experienced user to produce animated films of the execution of LISP programs. The fundamental concepts of LISP are seen as the S-expression and the process of evaluation (EVAL). ANTICS displays S-expressions in two ways, in the standard "pretty-printed" format or as "CONS" cells and pointers (a well established paper method of representation). The execution of LISP programs is carried out in terms of the operation "EVAL". The evaluation of the program consists of a split window display, the top half listing the user's code (in an internal representation, lambda notation), and the bottom half printing the call to the function and the code it calls (see figs 2.10 - 2.14).

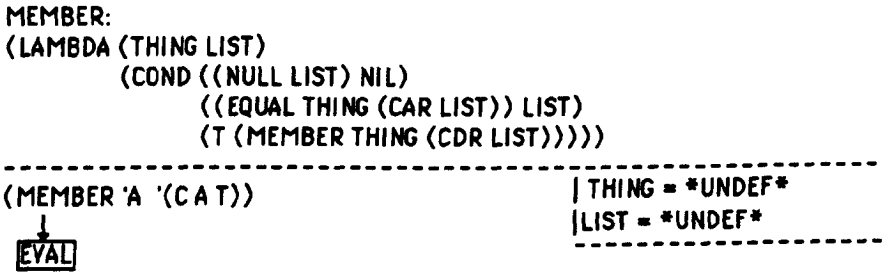



Figure 2.10 A series of screen snapshots from ANTICS

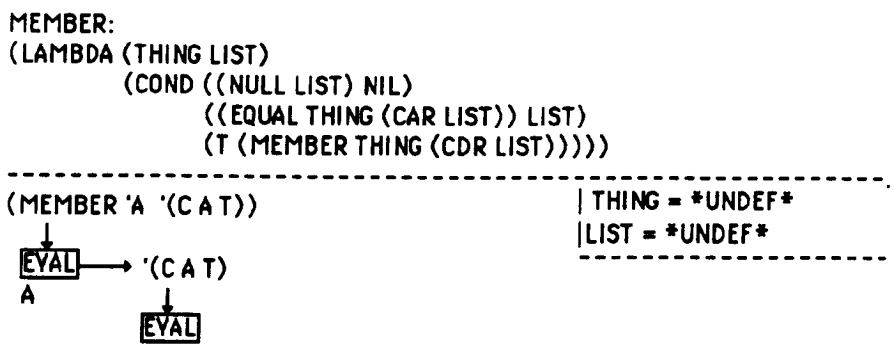



Figure 2.11

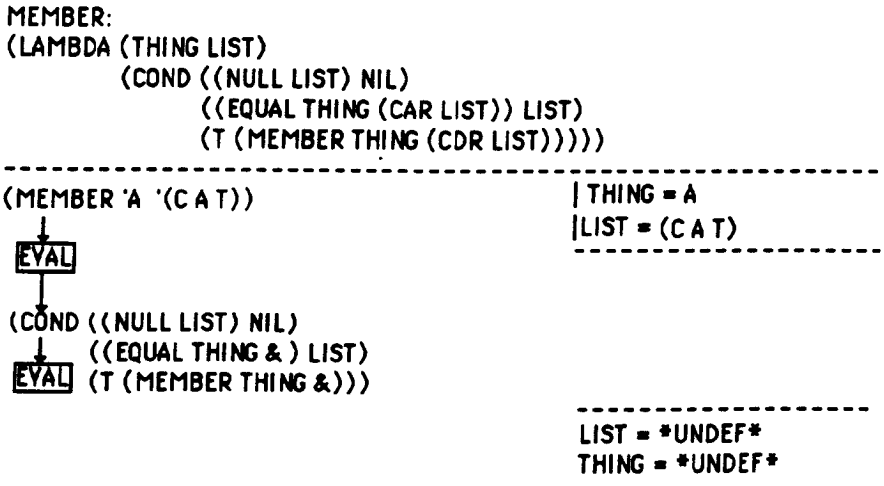



Figure 2.12

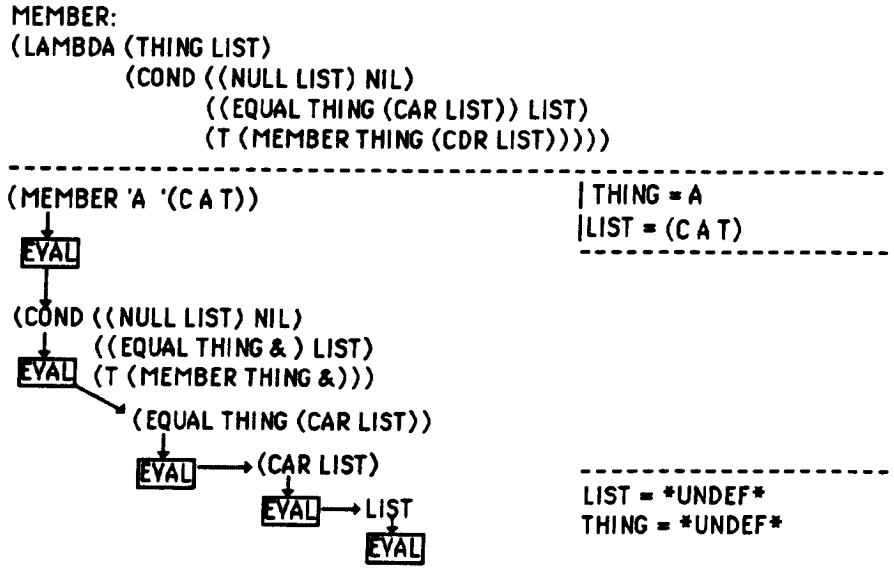



Figure 2.13

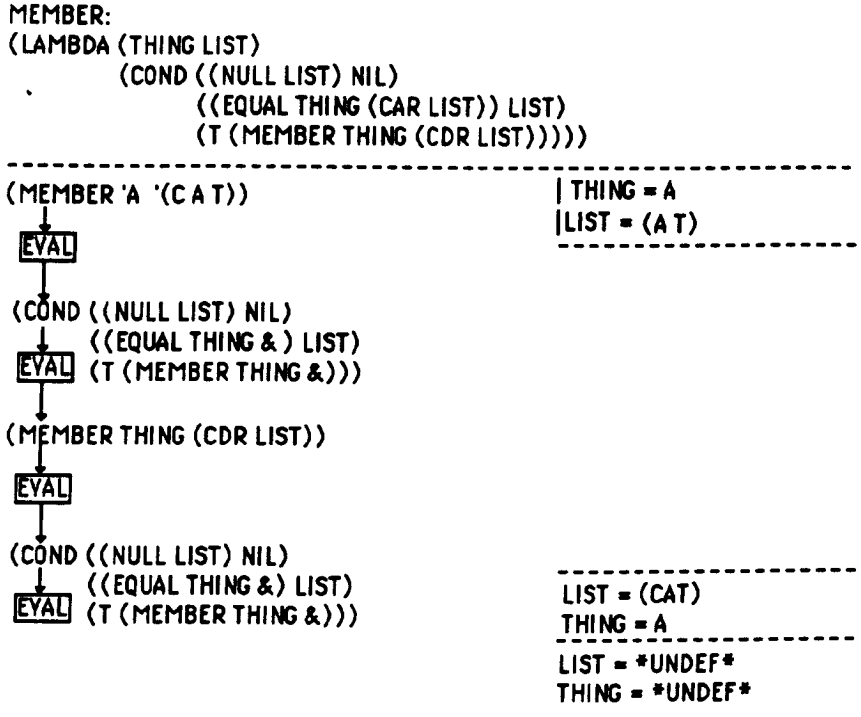



Figure 2.14

The system has two tracing modes. 'Stepmode' displays one frame at a time, and is under the user's control, while 'automatic' mode displays the trace at a rate which

can be set by the user. It is also possible to move backwards up the display and repeat the execution of a tricky piece of code.

The following are the rules used in the display:-

1) The S-expression being evaluated is displayed in the lower window, and below this is printed a box containing the word "EVAL". These are connected with an arrow. Screen space is saved and detail suppressed by limiting the printing of structures to a depth of three items.

2) If the form contains any arguments to be evaluated their evaluation is animated in turn to the right of the display (each with its own "EVAL" box), and the returned values are printed below the "EVAL" boxes.

3) The variables can optionally be displayed in the top right corner of the lower window, together with the variable bindings. This is updated as the bindings change.

4) Throughout the animated sequence the definition of a LISP function can optionally be displayed in the top window. The part of it being currently executed is intensified.

5) The window scrolls to accommodate new information from the top or bottom of the screen.

During the trace ANTICS shows the edit-time code as well as the sequential evaluation of the trace-time code, allowing the user to associate the static representation of the program s/he has written to the dynamic representation shown by the tracer. This association of edit-time to trace-time code is augmented by the intensification of associated pieces of program code in both windows when that piece of code is evaluated. The user may also move backwards and forwards in the trace, so that a tricky piece of code can be retraced without having to start all over again. Unfortunately for the novice, the edit-time code is presented in an internal notation which is different from the notation used by the novice when writing a program.

As in the previous system variable bindings at run time are displayed in isolation from the trace-time code in the corner of the lower window. If the run-time binding of variables was shown integrated into the trace-time code it would allow the novice to see the variables become bound in the context of the program, rather than search around the screen for the information.

Another intrusion on the user's concentration on program execution is the use of arrows and 'EVAL' boxes to represent the dynamic nature of the evaluation of each S-expression. If these characters were removed from the display it would allow the user to see clearly the information concerning program execution. This limit, of three items, the system places on the display of lists prevents the novice from seeing the whole picture of the execution of a piece of code, and may keep just the piece of information the novice wants to see off the screen. Novices will also have problems associating the code they have written with the displayed execution of that code with all these foreign symbols printed here, there and everywhere. The important tracing features can be shown by more powerful methods such as inverse video.

### *2.4.4 The Cornell Program Synthesizer*

The Cornell Program Synthesizer (Teitelbaum and Reps, 1981) is an integrated environment based on a syntax-directed editor and command templates for both Pascal and PL/1. In this system it is possible to trace the flow of execution through the program in a fairly basic way. When tracing takes place the screen cursor indicates the focus of attention in the source code as the program executes (see fig. 2.15). The stopping places for the cursor during tracing correspond to the structural units of the syntax-directed editor, one cursor jump for each template and phrase. When control passes outside the screen window the display is automatically redrawn to accommodate the new code. Code that the user finds uninteresting can be skipped over when tracing by inserting commands into the code. It is also possible to monitor selected variables during tracing, these and their values being displayed in a separate part of the screen. It is only possible to trace one element of an array at a time, however.

```
DO WHILE (k < n);
   IF ( k>0 )
      THEN PUT SKIP LIST ('the number k is strict
                  ly greater than zero');
      ELSE PUT SKIP LIST ('not positive');
   k= k + 1;
   END;
```

Figure 2.15 A screen snapshot from the Cornell Program Synthesizer

The trace has two modes: automatic and single step. In automatic mode the rate of the presentation of the display can be altered by the user, allowing a rapid movement through the trace. Single stepping provides a more sedate look at the execution of a program. The user can also move backwards through the trace to repeat part of the execution not understood.

This system has an integrated programming environment allowing the novice to use the tracing system immediately without learning a tracing specific environment and new set of commands. This will reduce the amount of knowledge the novice needs to know before s/he can start programming. It also leaves him/her to the task of concentrating on the view of program execution that is being presented rather than on the hows and whys of a new programming tool.

During the trace of a program the edit-time code can always be seen, as the focus of attention of the tracer is displayed in the edit-time code with the editing cursor. However the sequential evaluation of the edit-time code (what each piece of program does when it is executed) is not shown to the user. The values that the program's variables hold at run time are shown in a separate part of the screen to the edit-time code, and the variables that are displayed have to specified by the user beforehand. While the ability to specify which variables are to be traced would be of great benefit to an experienced user, being forced to specify the traced variables places an extra burden on the novice programmer.

### *2.4.5 Boxer*

Boxer (diSessa, 1982) is a programming language which has been designed as the basis of an integrated environment. It relies heavily on graphics, is based on LISP and LOGO, and is specifically aimed at naive and novice users. Boxer is a top-level editor and all communication with the computer is carried out through it. The editor is menu driven, the commands are executed by pointing at them with a mouse and hitting a 'doit' key.

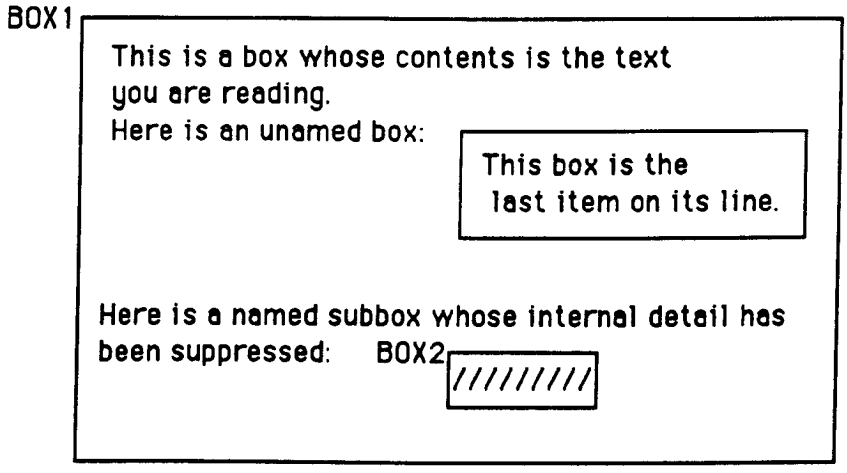



Figure 2.16 A series of screen snapshots from Boxer

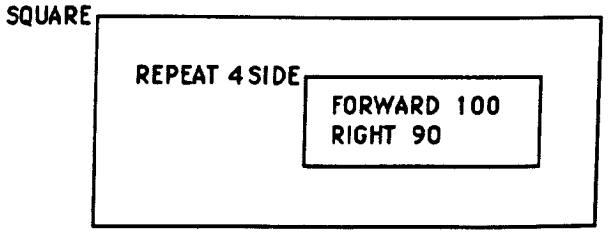



Figure 2.17

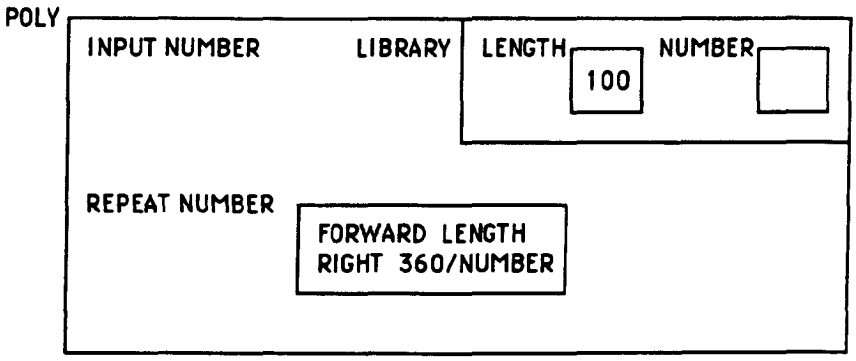



Figure 2.18

The basic structure in Boxer is the 'box' (see figs 2.16 - 2.18), boxes can be created, deleted and moved. A box is a rectangular window containing text, and possibly sub-boxes. Each box has a label, just like a procedure name. Procedures appear as boxes, which can be named and used as sub-procedures. The upper right hand corner of a box contains the 'local-library' which defines any local symbols used in that box or sub-box, and these can be given default values. Alternatively a box can represent an environment, such as a file directory, or a data object (strings, arrays, lists).

Debugging aids are proposed but not yet built, however the idea mooted in the system description is contained in the following extract:

> In particular, we wish to implement a method of watching a program in action to spot the error. Debugging, of course, is important in its own right for a host of other reasons. But perhaps most important, the visual method we've chosen to implement will aid the acquisition of the intended models as well as simply the catching of bugs. We expect episodes of watching the behaviour of the system to lead to a rich set of rationalizations and other partial understandings important to incremental learnability .[p26]

Boxer is trying to improve the ease of use of programming languages for novices by creating a new language based on features from LOGO and LISP, and on graphics

facilities. The idea of boxes lends itself well to displaying the structure of an environment or program, especially where concepts like recursion are concerned. The system described in DiSessa's paper is still in its early days with many features only proposed, so it remains to be seen what Boxer will develop into. The proposed tracing system comes to the same conclusion as that presented in chapter 2. The aim of the tracer is to alleviate the problems novice programmers encounter by allowing them to watch programs in action so that they may acquire a dynamic model of how the language works. This will allow the novice user to make predictions about the run time actions of particular language features.

### *2.4.6 MAGPIE*

MAGPIE (Delisle, Menicosy and Schwartz, 1984) is a programming environment for Pascal in which all the tools are based on a single and consistent user interface. It is not built primarily for novices, but is a flexible system more useful to the expert programmer. The system however has some ideas which will benefit the novice programmer and help him/her attain the knowledge to become an expert.

The display in MAGPIE consists of multiple windows called browsers (fig. 2.19), of which only one is active at any one time. This is denoted by the title bar of the window being highlighted. Commands are given via pop-up menus (the commands are consistent throughout the environment, always working in the same way and doing the same things) which contain commands only relevant to the contents of that window.

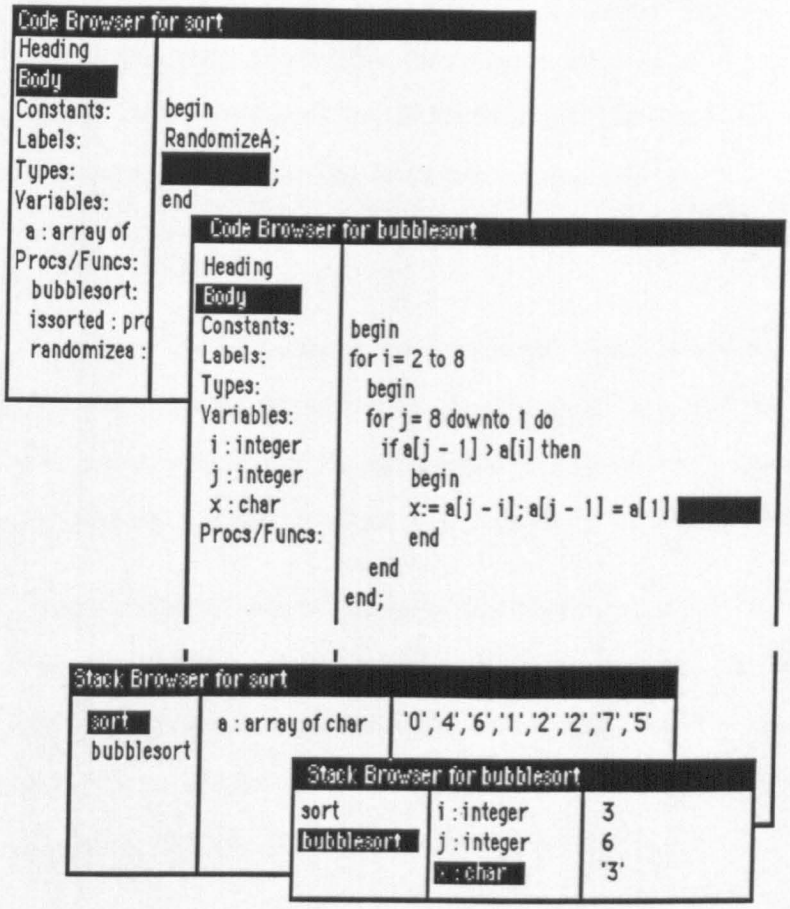



Figure 2.19 A screen snapshot from MAGPIE

Each browser has a number of panes, always set horizontally. The left hand pane always contains a list of categories and identifiers defined within each category, for example Heading, Constants, Labels, Variables. When moving through information contained in the panes the relevant category is highlighted. In the editor (code browser) there are two panes, the right-hand pane containing the user's program. In the stack browser there are three panes. The left-hand pane displays the stack of procedures and functions, the middle pane contains a list of variables, and the right-hand pane the values of those variables.

MAGPIE provides a trace of the execution of a program through these browsers via highlighting the edit-time code, the category, and showing the value of specified variables. This is the limit of MAGPIE's tracing capabilities, due to the following:

> Because extensive tracing slows down program execution considerably - due to the overhead required to update the display - the programmer will typically trace only the variables and procedures that are being debugged.[p55]

Novices do not have enough of a framework of knowledge about program execution to make decisions about which variables to trace. Showing variable binding and the sequence of program execution in an integrated display should allow the novice to build a framework, or conceptual model of the language.

MAGPIE has not been written especially for novices, and accordingly the power and flexibility of its approach would be too complex for them. However if MAGPIE'S environment could be simplified and made friendlier, and the flexibility of the system could be hidden from beginners until they became more adept at programming, then this approach might benefit novices.

MAGPIE traces programs by highlighting the edit-time code to show the order of execution of the program. This conveys the dynamic nature of the program, but does not show what happens to each piece of code when it gets executed. As in previous systems, with the same criticisms, only specified variable bindings are displayed to the user. The best feature of this programming environment from the novice's point of view is that all the tools are based on the same design. This means for all intents and purposes that there is only one tool, because they all look the same and work in the same way. This reduces the cognitive load imposed on novices by reducing the number of new rules that need to be learned before they can start learning a programming language.

These features only provide a very limited view of program execution, which will not give the novice sufficient information to build a conceptual model of what happens to programs at run time.

### *2.4.7 PECAN*

The PECAN system (Reiss, 1984a; 1984b) is similar to MAGPIE. It consists of a family of program development systems, which provide multiple views for algebraic programming languages, e.g. Pascal. It is stated that PECAN environments are designed for both novice and experienced programmers. This is a very flexible system with three types of editor (syntax-directed, declaration and structured flow graph editor) and several semantic views of the program, including expression trees, data type diagrams, flow graphs and symbol table. The system is accessed via menus and templates (see fig. 2.20).

PECAN provides several views of program execution, consisting of three types, control, program and data. The control view provides debugging facilities, and contains messages indicating the current state of the program. It also displays run-time error messages, program input and output. The data view shows a traditional display of the stack and variable bindings. This view is split into two parts, the left half shows the current execution stack, with variables and their current value. The right hand side is where the complex values, too large to fit in the left hand part of the data view, are shown, for example arrays. Program execution can be monitored in the program views, including the flow graph view which when active dispays the step by step execution of the program. Each active program view highlights the current statement as it is executed. The user can reverse program execution which allows stepping through the program both forwards or backwards. A speed control can be used to slow down or speed up program execution.

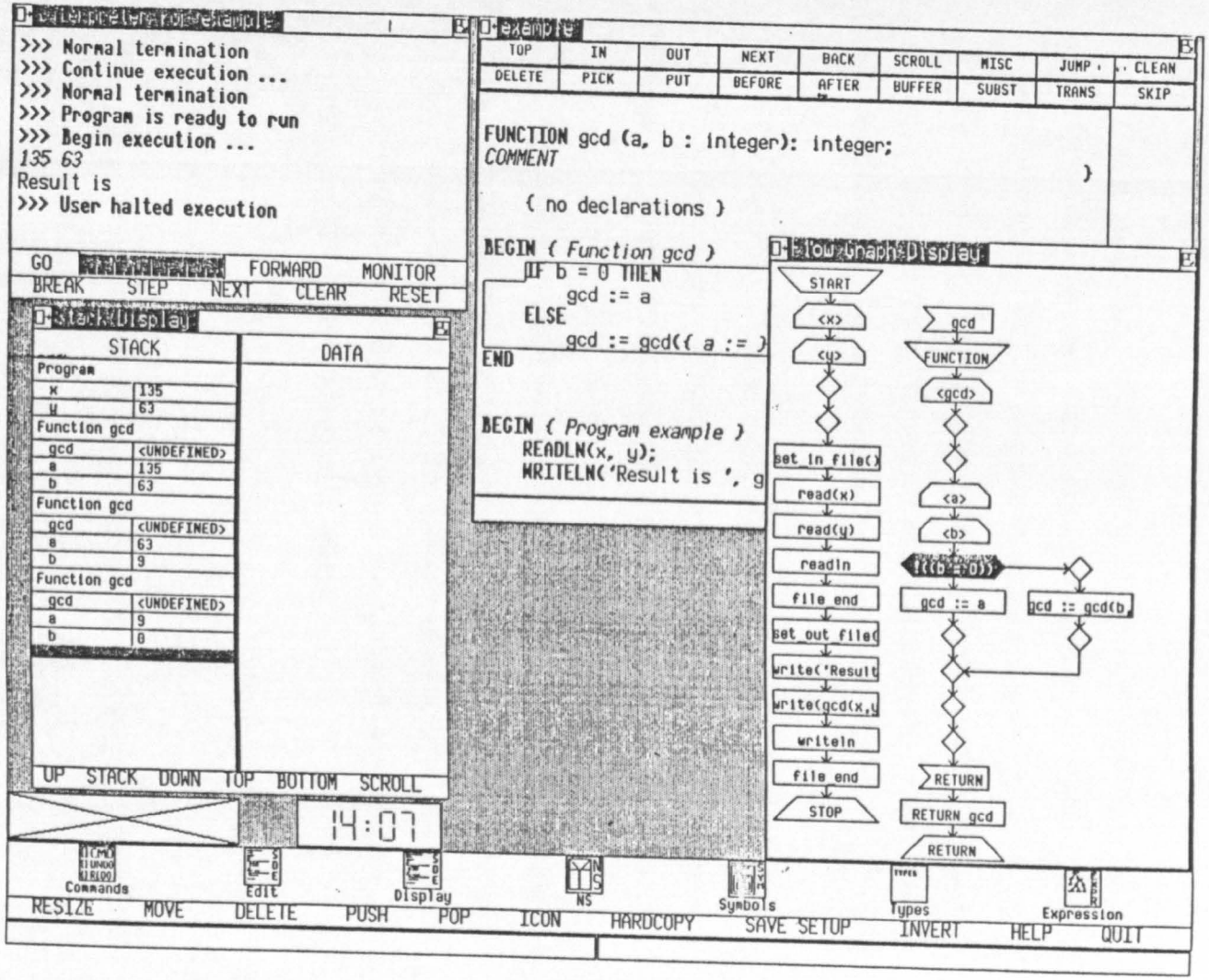



Figure 2.20

A Screen Snapshot from PECAN

Future improvements to viewing program execution include: a graphical representation of data such as a tree which dynamically updates as the program executes, and execution views of the program in action.

PECAN provides an excellent approach for the expert who can use his understanding of the underlying workings of a language to determine the meaning of the different views display. However, the separate views displaying: the sequence in which the program is executed; the binding of variables; and the input, output and run-time error messages of the program do not allow the novice to see a single integrated story of program execution.

The novice has to look in several different places, interpret the display, and then combine the views to understand what is happening to the program as it executes. In order to use PECAN the novice must learn how each of the views of program execution work and what they mean, which increases the start-up time of the system. These separate execution views do not allow the user to see both the edit-time code and an integrated view of the trace-time code. This would enable the novice to associate the static form of the edit-time code to the dynamic form of the trace-time code. It is possible that the future improvements to PECAN may answer these problems with its view of the program in action.

### *2.4.8 Zstep*

Zstep (Lieberman, 1984) is an interactive stepper for LISP developed on a dedicated LISP machine. The system integrates an advanced stepper with a real-time full screen editor, displaying both program and data (see figs 2.22 - 2.24). The control structure of the stepper allows the user to zoom in on a bug, by examining the program at a coarse level to start with and then at finer levels. Since it keeps a history of the stepped code Zstep can run both forwards and backwards.

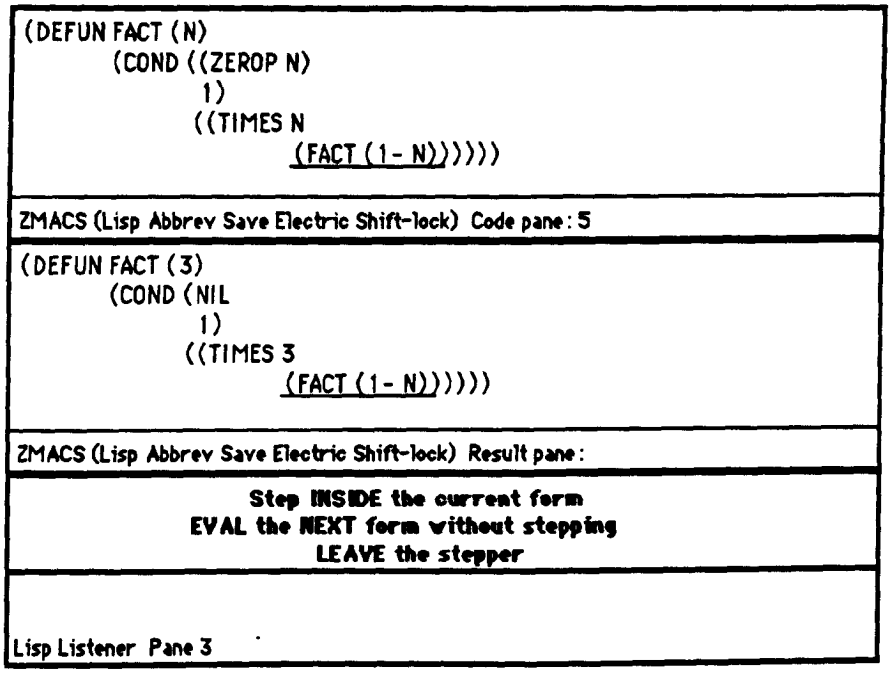



Figure 2.21 A series of screen snapshots from ZSTEP

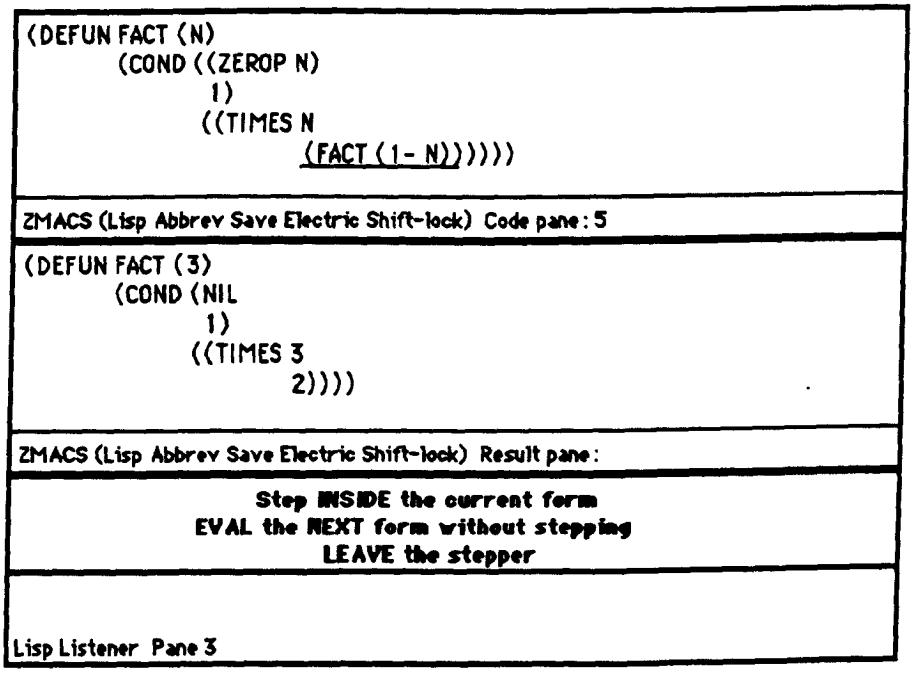



Figure 2.22

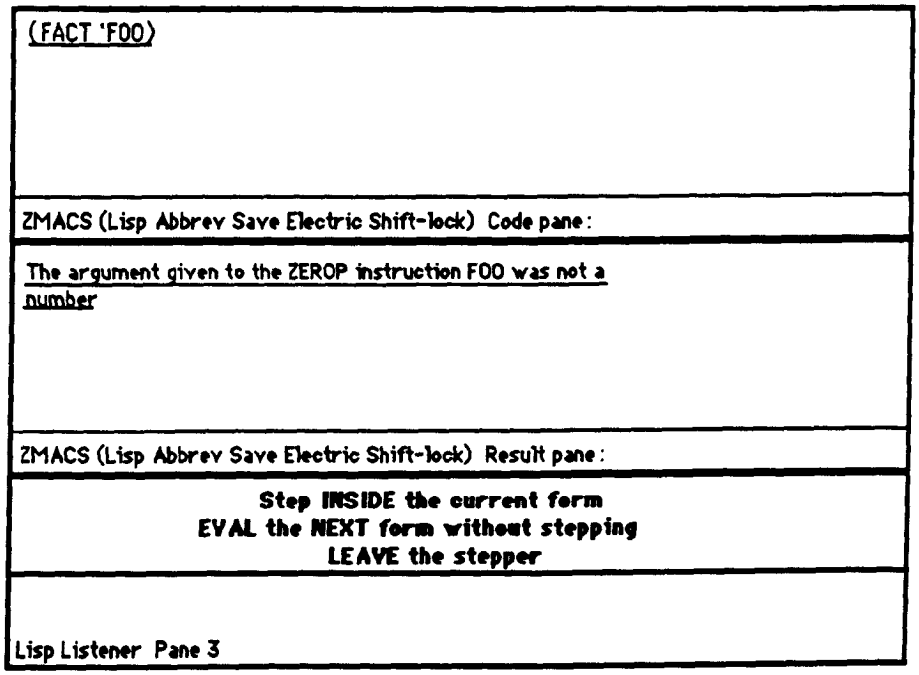



Figure 2.23

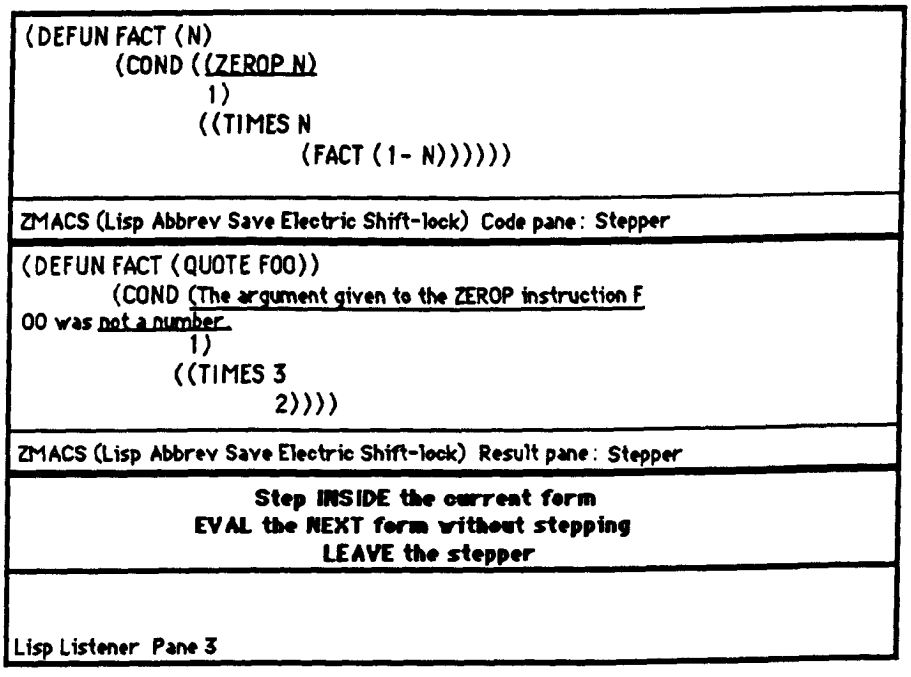



Figure 2.24

The display consists of four windows (the windows being set vertically), two editor windows, a command window and a top-level window. The top window, called the 'Code pane' shows the edit-time code which contains the LISP functions which have been written by the user. The next window, called the 'Results pane', displays the trace-time code and shows an animated view of the sequence of program execution.

Tracing starts with the function that has been called for evaluation being displayed in both text windows (see figure 2.21). First the s-expression for each of the function's parameters is replaced with its corresponding value from the function call. Then each sub-expression found in the function definition is evaluated, and the expression in the trace-time window is replaced by the value that it evaluates to in turn (when LISP functions are evaluated by the interpreter a value is returned). When each sub-expression is evaluated it is underlined in the edit-time code window. At the same time the trace-time window shows the underlined sub-expression being replaced on-screen by the value that expression evaluates to. After evaluation the value is left underlined and Zstep pauses before moving on. If at any point the code being stepped produces an error, the system replaces the error producing expression in the trace-time window with the interpreter's error message, which can be expanded if it is not fully understood.

The command window contains stepper commands such as a) step inside the current form (form is another term for s-expression), b) evaluate next form without stepping and c) leave the stepper. Zstep has an option of only displaying the trace-time window, which allows the user to view complex code more easily, instead of both the edit-time and trace-time code. It is also possible to move back through the trace to repeat the execution of a tricky piece of code, without having to trace the whole program again.

Summarizing: Zstep steps through Lisp programs simulating the way the Lisp interpreter evaluates the user's code. As the Lisp code is evaluated it is replaced on

screen by the result of that code's evaluation. A pure copy of the user's edit-time code is displayed in the window above the evaluation for reference. Variables in the 'trace-time' code are replaced with the values they are bound to, and when functions are called their definitions replace the calls on screen.

The trace ZSTEP provides is displayed in terms of the code the user has written, and shows both the edit-time and trace-time code. This means that the novice will be able to recognise the code being stepped as his/her program, and can see the relationship between the static edit-time code and the animated/dynamic actions of the trace-time code. Zstep provides a model of the evaluation process of LISP and presents a clear story of what is happening when a program runs. The trace displays the sequence of evaluation of the program, which is aided by the underlining of the s-expression currently under focus, and what each piece of code in the program evaluates to. All the aspects of evaluation are shown, including variable binding, expression and function evaluation. When errors occur in the user's program the error message produced by the interpreter replaces the code that produced the error in the trace-time code. This shows the novice exactly which piece of code has produced the error, because the error is given in the context of the program. The error message can be related to the user's program instead of being a general error message for that type of error. This should provide a concrete basis for the novice to build a hypothesis for debbuging, and therefore facilitate the understanding of the bug and speed error correction.

The only criticisms of ZSTEP are that in showing the evaluation sequence of LISP programs, it moves too fast for novices that are just starting to learn programming, i.e. the first week. It seems more suited to users who have passed this first hurdle and already have some notion of how LISP works. The commands that control the stepper, shown in the figures as a menu in the lower part of the screen, use terms that the novice will be unfamiliar with and describe actions that will not be known. For example 'form' is a term that is not normally found in text books until

after the first few chapters, and 'EVAL' is a complex concept that novices are unlikely to have grasped for some time let alone the first couple of days. ZSTEP is designed specifically for LISP and does not present a theory, for tracing programs in action, which is unfortunate as it contains many desirable features of benefit to novice programmers. Zstep would be more useful to users who already have some knowledge of how LISP works rather than to pure novices who are attempting to learn the basic concepts of the language.

### *2.4.9 PTP*

PTP (Eisenstadt 1984) is a Prolog Trace Package which gives a detailed retrospective view of program execution. PTP is based on the 'Byrd' box model of Prolog (see section 2.4.10) with the addition of information concerning goal selection and satisfaction.

The 'Byrd' box model of Prolog only provides execution information telling the user that a goal has either succeeded, failed, exits or is being retried. PTP provides the user with extra detail showing why a goal has succeeded or failed. This information is based on the following view of Prolog program execution:

> Goals may succeed because they are trivially true (facts) or because they have subgoals which succeed. Goals may fail (1) because subgoals have failed and there are no resolving clause heads left; (2) because a subgoal has failed back to a cut which by definition fails the parent goal; or (3) because no clause head can possibly resolve against this particular goal. The latter fault in turn may have one of the following causes: (3a) no definition of the relevant predicate exists at all; (3b) the existing definition is of a different airity (number of arguments) from that of the attempted goal; (3c) variable instantiation has failed (i.e. variables do not unify).[p516]

PTP uses 19 symbols, as flags, to represent the events in the above model, a sample of which are shown below:

| Symbol | Example | Meaning |
|---|---|---|
| ? | ? g(X) | About to attempt new goal |
| > | > g(foo) | Entering the body of resolving clause |
| } | } write(a) | System primitive |
| + | + g(foo) | Success: subgoals succeeded |
| +* | +*g(foo) | Success: fact in database |
| ++ | ++write(a) | Success: system primitive |

figure 2.25 Some of the symbols used in PTP

Likewise there are symbols for six different types of failure; attempted unification of clauses; encountering the cut, and backtracking to cut.

PTP runs in either a one window or a two window mode. The one window mode allows the user to see a large amount of the traced code, while the two window display shows the source code in the top window and the trace in the lower window (see figure 2.26).

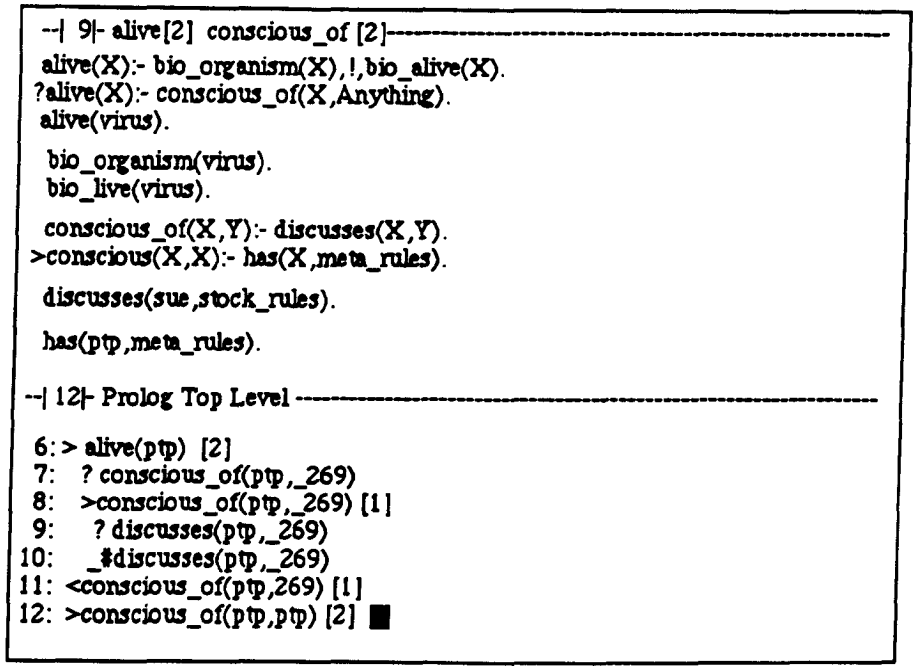



Figure 2.26 A screen snapshot from PTP

The top window consists of a status line and a scrolling region which shows the source code which is relevant to the trace. The status line, at the top of the screen, shows an 'invocation hierarchy trail' which scrolls horizontally, the right most clause being the current goal. Within the source code the system shows the user: which clause has resolved against the current goal (flagged by '>'), and the parent goal of the current goal (flagged by '?'). These flags are dynamically updated as the trace proceeds.

The lower window displays the same information as in the one window trace mode, described below.

The one window display attempts to make all the aspects of execution explicit. The display shows the four aspects of the 'Byrd' box model, i.e. whether goal succeeds, fails, exits or is retried. The display (see the lower window of fig. 2.26) consists of a line number for reference; a flag saying what has happened to the clause

which follows the flag; and if this clause has been resolved, the number of the clause in the database referenced by predicate name. The clause shows the value of variables if they have been instantiated, or the internal number for compatability with other Prolog trace systems.

In addition to the four aspects of execution shown by the 'Byrd box' model PTP gives the information concerning why a goal failed or succeeded. PTP uses different flags to represent the different types of success and failure.

To prevent the user from being overwhelmed by the amount of information presented by tracing systems, PTP presents an overview of execution to start with. This consists of information concerning the four ports of the 'Byrd' box model. If the user wishes s/he can request a more detailed view of the trace. This can take the form of a request either, to see everything that happens in the entire trace, or to zoom in on the detail of what happens to a particular goal (referenced by line number). It can do this because the trace is retrospective, and all the tracing information is stored internally. This allows the user to see a global view of program execution and then zoom in, in successively greater detail on interesting subgoals.

In addition to the tracing PTP also provides facilities which analyse the trace and provide the user with plausible reasons why a bug occurred. 'Suspects' displays PTP's analysis of the trace (performed at the time the trace-time information is being stored) identifying the following suspect cases which cause errors: missing definition; wrong airity, and non-resolved goals. 'Explain' takes one argument, the line number of a failed goal in the trace, and attempts to explain why this goal has failed. It does this by invoking a specialist which looks for standard failure patterns (cliche) in the internally stored trace. An English explanation then tells the user the reason for failure. In the end the user must decide why the bug has arisen, but now s/he can make an hypothesis based on the information provided by PTP.

This system provides much of the information concerning program execution that would allow novice programmers to see what happens to a program at run-time. However, the presentation of this detailed information is aimed at expert users rather than novices. The flexibility of PTP in showing the detail of execution required by the user allows the expert to move quickly through the trace, zooming in on the bug which they can see in detail. In addition to this they can ask for help in the analysis of the trace to suggest possible causes for the bug.

This approach is directed by the user to home in on the bug, which is interpreted with the help of the suggested suspects. To understand the display the user must have learned the meaning of the 19 symbols that are used to represent the different aspects of program execution in Prolog. These symbols increase the amount of information that the user has to know in order to use PTP. In other words it means that the novice is trying to learn how Prolog works and what the symbols mean.

If the symbols were replaced with a brief English explanation on a status line this would eliminate the need for the novice to learn 19 symbols. The extra detail in the English explanation would clarify the view of program explanation.

The presentation of both the source code and the trace-time code (in the two window display) allows the user to associate the static form of the progam which has been edited to the dynamic form of the program at run-time. However, because the trace-time code only shows discrete parts of the execution process, the user does not see how the execution proceeds from one step to the next. This procedure is implicit in the meaning of the 'flags' rather than explicit in the display. If it was shown it would allow the user to explicitly see what is happening all the time, and mean that they have to rely less on the flags or status line for execution information. This extra detail would show *how* unification and varible instantiation take place, rather then just see the result of these processes

### *2.4.10 SODA*

SODA (Plummer, 1985) the Screen Oriented Debugging Aid, is a screen oriented approach to presenting the execution of Prolog programs. It is based on the 'Byrd' box model of Prolog execution (Byrd, 1980) which displays the occurance of the four things that can happen to a Prolog goal at run time, 'Call', 'Exit', 'Redo' and 'Fail' (Fig. 2.27). When a Prolog program is traced the 'Byrd' box model tells the user when a goal is 'Call'ed, 'Redo'ne, 'Fail'ed, and 'Exit'ed, including the values of any variables that are instantiated at that point. Uninstantiated variables are shown as internal adresses, and provide little information to the user apart from the fact that a variable exists that is unbound, it is difficult to tell which variable is unbound because no variable name is displayed for reference. An example of a 'Spy' trace which is based on the 'Byrd' box model is given in chapter 1.

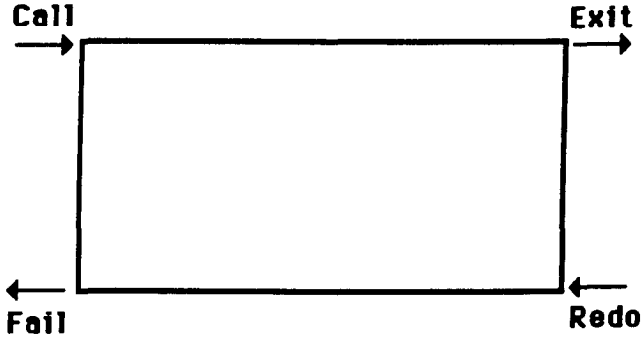



Figure 2.27 The 'Byrd Box' model of Prolog program execution

The model of the Prolog interpreter that SODA presents extends the 'Byrd' box model by adding a precursor (Fig 2.28). This precursor makes explicit the events that take place in order to determine which goal gets 'Call'ed.

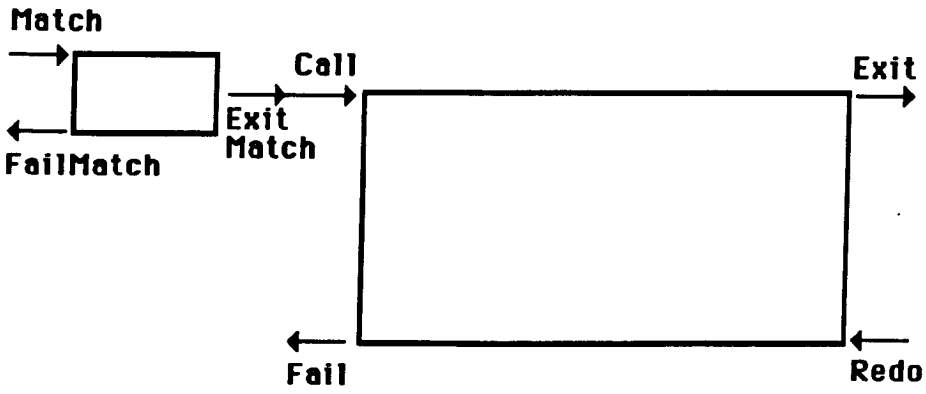



Figure 2.28 The SODA model of program execution

The box that SODA adds to the 'Byrd' model consists of three events. These events are as follows; the attempted match of a goal against database entries, the failure of such matches, and lastly the exiting of matches which denotes the success of a match. SODA shows the following run-time actions of the Prolog interpreter:

* Finds the first clause for the procedure being called,
* Matches the head of this clause with the call,
* If the match goes through then invoke the body, otherwiswe find the next clause.
* If the invocation of the body fails, find the next clause.

This process displays the order that Prolog searches through the database when attempting to unify goals. In the 'Byrd' box model this process is kept hidden from the user.

SODA is based on the standard DEC-10 Prolog debugger and so includes most of the commands normally found which allow the expert user to move around the trace at will, seeing some things in detail while skipping others entirely.

The following extracts from Plummer describe what SODA does when tracing a Prolog program (*my comments in italics*):

> The use of the extra ports is best shown by example. Suppose that we have clauses:
>
> ```
> foo([],_[]) :- !.
> foo([H|T],N,[NH|NT]) :-
>         M is N - 1,
>         baz(H,NH),
>         foo(T,M,NT).
>
> baz(b,a).
> ```
>
> When given the call "foo([a,b,c],4,K)." The first thing that you would see (*in a trace given by the 'Byrd' box model*) would be:
>
> ```
> (1) 0 Call _1 is 4-1
> ```
>
> The fact that there was a clause that didn't match is not made evident. When Soda is given the same call, it picks up all of the clauses for foo/3. and displays the goal and the first clause:
>
> ```
> foo([a,b,c],4,A)
>
> *>  foo([],_[]).
> ```
>
> The *> marker pointing to the head of the clause indicates that we are at the "Match" port of the debugger - about to attempt to match the goal with the head of this clause. After an input of either "creep" (*show the next step in the stepper*) or "skip" (*skip over the next step in the stepper* ) the next thing that would be displayed would be:
>
> ```
> foo([a,b,c],4,A)
>
> <*  foo([],_,[]).
> ```
>
> The <* marker indicating that the match failed and that we are at the "FailMatch" port. Another input of "creep" or "skip" will cause Soda to desplay:
>
> ```
> foo([a,b,c],4,A)
>
> *>  foo([B|C],D,[E|F]) :-
>         G is D - 1,
>         baz(B,E),
>         foo(C,G,F).
> ```
>
> We are at the "Match" port of the second clause. So an input of skip in this context (*meaning do not show the detail of this call*) will result in the display:
>
> ```
> foo([a,b,c],4,[A|B])
>
> *>  foo([a,b,c],4,[A|B])  *> :-
>         C is 4 - 1,
>         baz(a,A),
> ```

```
foo([b,c],C,B).
```

Notice that the effect of the match has been propagated through the clause and the goal, and that the marker *> points away from the head of the clause indicating that we are at the exit match port.

An input of "creep" would have had a different effect. Creep in the standard debugger means: Take a single step, and then ask me what to do. Similarly in Soda; and at the match port this has the effect of matching the next argument and leaving a markerpointing at the next argument to be matched:

```
foo([a,b,c],4,A)

*> foo([a,b,c], *> B,[C|D]) :-
       E is B - 1,
       baz(a,C),
       foo([b,c],E,D).
```

Notice that the effect of the match so far has been propagated through the clause.

The Call, Exit, Redo and Fail ports behave exactly as they do in the standard debugger. Call and Exit are represented by the marker =>, appearing either pointing at the goal if the port is Call, and away from it if the port is Exit. Similarly Redo and Fail are represented by the marker <+, pointing at the baz/2.[p4-5]

This example gives a flavor of what the SODA system shows the user when tracing Prolog programs. I will discuss SODA from the point of view of the most detailed display it provides, i.e. the "creep", which would be how a novice would use it. This system displays the execution sequence of Prolog programs in terms of the user's edit-time code. However when the program is traced the system does not display the edit-time code, only the trace-time code. This means that when the database entries are matched against the current goal the user cannot see the other clauses that might match the goal. So the context of the database is lost thus losing the important information of why particular clauses are chosen from the database. It also means that users will not see the relationship between the static edit-time code and the dynamic trace-time code.

Only the current goal and its subgoals are shown on screen at any one particular time. This makes it difficult to see the context of the goal within the program because

the outstanding and previous goals cannot be seen, so the novice may get lost not knowing where s/he is in terms of the original program.

As the author mentions,

> the variable names that are displayed change as instantiations are made[p5]

and though as he says this might not bother experienced users it would confuse novices terribly, because one of the problems novices have is that of seeing how variables get bound and this means that they need to be able to follow variables throughout the trace. Another problem novices will have following the variable instantiations is that when variables get their values they just get replaced on screen without being explicitly pointed out to the user.

The movement through SODA's model of program execution is pointed out to the user with the characters, '*>'; '<*', '=>', '<+', which denote different instantiation/unification events that occur during run time. Although these symbols relate to the detail of instantiation/unification and may help the novice see what is happening, the user still needs to know what the symbols mean and may be distracted from the sequence of execution because the characters break up the trace-time code. Novices may find it difficult to understand the model behind this tracer, because at this stage the novice is attempting to build a working model of Prolog.

### *2.4.11 Summary*

In order to clarify the features of the above systems which are deemed either to be of benefit to, or a disadvantage to novice progammers I will present a summary below. The features of benefit to novices are shown in italics, while the disadvantageous features remain in plain text.

**Baeker**

*Execution sequence displayed*

*Trace shown in terms of the edit-time code*
*Process of variable instantiation/procedure calling shown*

not interactive
Restricted to show particular programming language features

**BIP**

*Execution sequence displayed*
*Trace shown in terms of the edit-time code*

Uses extraneous symbols to flag trace features
Limited tracing of variables
Variable bindings shown separately from the trace-time code

**ANTICS**

*Execution sequence displayed*
*Trace shown in terms of the edit-time code*
*Process of variable instantiation/procedure calling shown*
*Edit-time and trace-time code displayed together*
*Trace runs forwards and backwards*

Uses extraneous symbols to flag trace features
Limited tracing of variables
Variable bindings shown separately from the trace-time code

**Cornell Program Synthesizer**

*Execution sequence displayed*
*Trace shown in terms of the edit-time code*
*Trace runs forwards and backwards*
*Unified programming environment*

Limited tracing of variables
Variable bindings shown separately from the trace-time code

**Boxer**

*Unified programming environment*
*Proposed tracer wil allow user to watch the program being executed in detail*

**MAGPIE**

*Execution sequence displayed*
*Trace shown in terms of the edit-time code*
*Unified programming environment*
*Use of inverse video to flag trace features*

Limited tracing of variables
Variable bindings shown separately from the trace-time code
Flexible approach

**PECAN**

*Execution sequence displayed*
*Trace shown in terms of the edit-time code*
*Trace runs forwards and backwards*

Variable bindings shown separately from the trace-time code
Flexible approach

**Zstep**

*Execution sequence displayed*
*Trace shown in terms of the edit-time code*
*Edit-time and trace-time code displayed together*
*Process of variable instantiation/procedure calling shown*
*Trace-time code associated to edit-time code*
*Unified programming environment*
*Error messages displayed in the context of the trace-time code*
*Error messages associated to the edit-time code*
*Trace runs forwards and backwards*

Control of stepper complex

**PTP**

*Execution sequence displayed*
*Edit-time and trace-time code displayed together*
*Trace-time code associated to edit-time code*

Uses extraneous symbols to flag trace features

**SODA**

*Execution sequence displayed*
*Trace in terms of edit-time code*
*Process of variable instantiation calling shown*

Uses extraneous symbols to flag trace features
Variable names change throughout display
Control of stepper complex

## 2.5 Conclusion

It is clear from the above descriptions that the appearance in the past few years of improved graphics hardware and software (Symbolics™ 3600 , Macintosh™ and Lisa™, Xerox™ 1100) has enabled designers to implement much improved dynamic programming tools. However none of the above systems solves the problem of principles for the design for dynamic aids which may be used for any programming language. The systems have been built for specific programming languages, to show specific features, and although they have many useful and interesting features they do not help future designs of dynamic programming tools. This thesis hopes to advance

animated tracing tools by presenting a set of design principles for designers of this type of tool.

In section 2.2 I concluded that the problems that novices have when learning a programming language appears to stem from the fact that the 'notional machine' they are attempting to learn is dynamic in nature while the teaching materials novices have available to them, in terms of manuals, computing tools and tutorials, present the 'notional machine' in a static fashion. This problem manifests itself by causing novices problems in understanding such dynamic concepts as:

- flow of control.
- variable binding.
- recursion/iteration.
- side effects.
- other miscellaneous language specific concepts i.e. backtracking, cut.

To elucidate these dynamic concepts and other programming techniques it was proposed that the novice should be provided with a clear view of the action of the languages interpreter/compiler on user's programs. The systems in section 2.4 have attempted to provide this view in some way or other for various different programming languages, and have succeeded in this goal to varying degrees.

To provide a clear and informative view of program execution it appears important that the user is shown the order in which the program is executed and what happens to each part of that program as execution happens. This will demonstrate the dynamic nature of the user's program at run time. So that the user can recognise this sequential execution of his/her program the program should be represented in terms of the user's edit-time code, rather than being based on an internal representation of that program.

In order for the user to be able relate the dynamic action of his/her program, shown in the trace-time code, to the edit-time code they have just written both the edit

time and trace time code should be visible at the same time. To strengthen the association between the two forms of the user's program, i.e.static and dynamic, as each part of the program is executed the relevant piece of code should be highlighted in both the edit-time and trace-time code.

The trace display should contain enough detail to show why, how, and when things happen, for example variable bindings, function calls, goal matching. The trace-time code should show the current, outstanding and previous states of the program, so that the user can see the context of the current focus of the tracer. This focus of attention to the current action of the tracer should not be displayed by symbols, because not only do they clutter up the display but they must be learned by the user before the tracer can be used. Also if these symbols are depicting features of an underlying model of program execution there is the added problem for the user of knowing and understanding this model and the relationship between the model and the displayed symbols.

Another way of reducing the amount of information that the user needs to know before being able to use a tracing system is to integrate the tracing system into the other parts of the programming environment, for example the editing system. As the user will need to be able to use the editing system in order to program, it seems sensible to integrate the tracer with the editor so that the user will not need to learn any new commands to use the tracer, and will recognise the tracing environment as being similar to the editor.

All of the desirable features listed above combine to provide the novice programmer with a clear view of the 'notional machine' in action, rather than the static or snapshot picture that has been traditionally taught. The 'notional machine' can now become concrete, dynamic and visible to the user, showing in detail the dynamic concepts (flow of control, variable binding) that otherwise would cause problems to novices when learning a programming language.

The features which seem most likely to help in reducing these errors have been extracted from the animation systems, combined with the design principles from the system design literature and are presented in the next chapter as a basis for building animated tracing tools for novice programmers.

# CHAPTER 3

# DESIGN PRINCIPLES FOR A STORY OF PROGRAM EXECUTION

Section 2.1 showed that novice programmers need help to build a working model of the language they are learning. It proposed that one means of doing this is to present the 'notional machine' of that language in terms of an animated view of program execution

This chapter presents a set of design principles for such a view of program execution. The principles, which are drawn from the information presented in sections 2.3 and 2.4, are aimed at producing a view of program execution that will enable novice programmers to clearly see enough detail concerning the execution of programs in order to build a working model of that language.

Each principle has been given a mnemonic name to enable the concept it describes to be discussed easily. Following the name each principle is discussed in terms of what it offers to the novice programmer, and where it has been derived from.

## 3.1 General Design Principles

The 'a priori' principles underlying the design of an animated stepper are those of simplicity, consistency and transparency (du Boulay, O'Shea and Monk, 1981). These three principles present global guidelines which enables novices to easily understand, learn and use a computer system.

**a) Simplicity** Simplicity refers to a system having a small number of parts which can be explained by a small number of descriptions, and interact in a way that is easily understood by the novice. This attempts to reduce the amount of information the novice has to learn before s/he can use the system. Instead of learning how to use the system, the novice can now concentrate on learning about the items that the system manipulates. In this case a programming language.

To simplify a view of program execution two things can be done. Firstly, the command structure through which the user controls the tracer should be rigid (as opposed to flexible) and contain as few commands as possible. For example, the three commands to 'step', 'stop stepping', and 'restep' are sufficient to allow a novice user to carry out all his/her desired tracing actions. Secondly, the method by which program execution is displayed should be carried out by the minimum of transactions.

This means that with only a handful of information the novice user can control and understand the animated tracing tool which is presenting him/her with a view of program execution.

**a) Consistency** The method by which information concerning program execution is presented to the user (by way of the interface) should be consistent throughout the system. This means that if a display method is used in one place to describe an event, it should also be used elsewhere in the system, allowing the novice to transfer information learned in one part of the system to other parts in order to understand what is happening.

Together with the principle of 'simplicity' this means that the novice only has to learn the meaning of a small number of display methods, which describe the execution of programs, and always mean the same thing wherever they occur. This all helps to reduce the amount of work the user has to do in order to understand the information presented by the system.

**a) Transparency** Transparency means viewing selected parts and processes of the 'virtual machine' in action, by means of pictures and a particular story line. In order for novices to be able to build a working model of a programming language it is essential that they can see everything that happens during program execution. Many of the events that occur at run-time are dynamic so the execution view should be presented in a dynamic fashion allowing the novice to see the process by which these events take place.

A problem that crops up here is how transparent should the system be. Programming languages can be explained at various different levels of detail from the commands of the language to the way that language is implemented. It is important that the correct level of description is chosen for the user of the system. This is discussed in detail in section 3.2.4.

## 3.2 Specific Design Principles

The general principles described above give global guidelines concerning the design of a tracing tool which presents a view of program execution. This section presents principles which discuss specific points in the design of animated tracing tools. These principles describe in detail how the aims of these general principles may be achieved.

### *3.2.1 Edit-time and Trace-time Code Isomorphism*

The code that is used to show the run-time trace should be a direct copy that the user has typed into the editor. In other words the trace-time animation of program execution will be built from the user's edit-time code. This means that the novice should immediately recognise the structure and meaning of the code being traced. Most of the systems described in section 2.4 show program execition in terms of the edit-time code.

Both the edit-time and trace-time code should be shown together during the trace, so that the events which occur in the trace-time code can be related to the edit-time code. This will enable the novice to see the relationship between the dynamic form of the run-time code to the static form of the code written in the editor. It should also help the novice develop future programs, enabling him/her to predict what will happen to the edit-time code at run-time. The display of both the edit and trace-time code can be seen in ANTICS, ZSTEP and PTP (section 2.4)

### *3.2.2 In-place Subroutine Instantiation*

This refers to the insertion into the trace-time code of any called piece of code, whether it be a procedure call in LISP or a goal call in Prolog. This will build up the animation of the execution of the user's program by adding new code to the trace as and when it is required. In terms of program execution it should help the novice assimilate a model of how flow of control works in the programming language. In the same manner if code has to be removed from the trace-time code, for instance in backtracking and the cut in Prolog, or function evaluation in LISP, the sequence of execution should be displayed explicitly. Both ANTICS and ZSTEP (section 2.4) show this feature.

### *3.2.3 WYSIWHa - 'What You See Is What Happens'*

What you see is what happens' is a new idea of presenting the 'virtual machine' in operation, giving novices a concrete view of program execution which is normally taught as an abstract concept. This not only aids the previous principle in showing how flow of control works, but shows exactly what happens to that code when it is evaluated thus providing a clear view of program execution. In essence this is making the view of program execution 'transparent' to the user (mentioned in section 3.1), allowing him/her to see the 'notional machine' in action. This idea can be split into the following principles:

**a) Fidelity to the true evaluation sequence of the code.** The instructions in the user's code are executed by the stepper in the same order as they would be executed by the interpreter. This provides a one to one mapping between what the user sees and what the interpreter is doing, hopefully preventing the user from picking up any misconceptions about program execution which can happen when analogies are used to convey information (Halesz and Moran - section 2.2).

**b) Side effect visibility.** 'Side effect visibility' should allow the user to see explicitly any side effects that the traced program makes to the programming environment. This might happen because of the use of global variables in LISP (and many other languages), or when an 'assert' is carried out at run-time in Prolog. Normally the user would not see side effects happening at run-time and can therefore be blind and confused by the effects they have on the rest of the program.

**c) Integration of variable instantiation within the trace-time code.** The process by which variables get bound to their values should be displayed to the user in the context of the trace-time code. This means that when variables in the trace-time code are evaluated they get replaced with a value. This enables the novice to see how, why and when each variable in the program gets bound to its value. This integration of variable instantiation into the display of program execution means that the novice only has to look at one view to see what is happening, rather than several views if the variable values are shown seperately.

The integration of variable evaluation in the trace-time code can be seen in Baeker, ANTICS, ZSTEP and SODA.

### *3.2.4 Description Level of Trace*

This principle is concerned with the description level of the story of program execution presented to novice programmers. At first thought it seems obvious to tell the user the truth about program execution. However, a problem arises when you ask the questions, "What is the truth?", and "How does the language work?".

Any programming language usually has many different implementations, some written in machine code, others in Assembler, while still others will be written in higher level languages like 'C' or Pascal. This means that at the level of the implementation language, the programming languge will work in many different ways. However at a slightly higher level all implementations of a programming languge will work in the same way. This behaviour is normally determined by a specification of what the language is meant to do, and how it is meant to do it. So, the way a language works, and the behaviour of its model is dependent upon what level of description is used to describe its actions.

So the new question becomes, "What level of description should be used in a display of program execution?". The level of description used is ultimately determined by the end-user of the system which is being built. If the end-user is working in the field of language implementation he/she will want a model of the language with a very low level of description, i.e. one that displays all the implementation-dependent features of that language. On the other hand if the end-user is a novice programmer, which is the case in this thesis, then the model of the language presented should be independent of the specific implementation. The level of description, or 'the truth', should be the outward appearance of the language which is stated in the language specification. This should ensure that the behaviour shown by a story of program execution is consistent, and works in all the situations that the user will come across. The aim of this 'truth' is for the user to build a useful model of the language, which can be used to interact and predict the actions of that language.

Another problem on a similar theme is that as soon as a system that models the behaviour of an environment is added to that environment, then the intrusion of that system on the environment changes its behaviour. To prevent the behaviour of the environment being changed enough to require another model, it is necessary to design the system so that it has a minimal affect on the environment.

### *3.2.5 Status Line Navigation*

'Status Line Navigation' is a method by which the user is kept informed of what is happening in the animated trace at all times. This idea has been used in some single-user workstations such as those from Symbolics™ and Perq™ to provide the user with information concerning the percentage of tasks carried out; the function of mouse buttons; and disk access.

The status line should provide a continual brief commentary on the current state of the stepper in order to clarify the stepper's action and to prevent the user from getting lost. The messages will comment upon events such as variable instantiation, subroutine instantiation, input and output, and evaluation. The presence of a status line reduces the need for symbols to point out the important features being displayed by the stepper. This leaves the screen clear for displaying the text of the program.

The status line will in effect be presenting the user with an English version of a model of program execution. This means that unlike traditional tracers the user will not have to learn a model of program execution before being able to use the tracer. The novice can thus use the tracer immediately to help build a working model of program execution.

Having this type of information contained in a status line means that if the user understands what the display is showing s/he can ignore the status line completely, and just watch the animation. This is not possible if symbols are used to flag tracing features, as they are still present within the trace-time code.

### *3.2.6 Trace Forwards and Backwards*

PECAN, The Cornell Program Synthesizer, ANTICS and ZSTEP all allow the user to run the view of program execution both forwards and backwards. This is an important facility for novice programmers as it provides them with an easy way of reviewing the execution of a tricky piece of code. It is likely that novice programmers will find many pieces of code tricky and will therefore wish to see the execution of that code several times before being satisfied that they understands how it works. Unless a facility for reviewing program execution is provided the novice will have to reinvoke the tracer over and over again. Not only is this time consuming, but it interrupts the user's train of thought.

### *3.2.7 Integration of the Interpreter's Error Messages*

This principle has been drawn from ZSTEP (section 2.4.8), and aims to provide the novice with contextual information concerning the cause of error messages in the same way as the principle discussed in section 3.2.2 concerning variable binding does for variable instantiation.

The error messages given by the interpreter should be integrated into the trace-time code so that the user can see the context of the process leading up to the error. If the relevant piece of code is highlighted in the edit-time code this will allow the user to see exactly which piece of code has produced the error, and provide a starting point for error correction. The error message should be as specific to the user's code as possible, for example saying which variable is at fault, rather than being a general message that is presented for a particular type of error. An expansion of the error message should be available in case the user does not understand the initial message.

### *3.2.8 Uniformity of the Editor, Top-level and Utilities*

The tracing system should be integrated with other systems such as the editor so that the command set and format of the display are the same throughout the environment. This is an attempt to make the programming environment as 'simple' and 'consistent' as possible at a global level. This should enable the user to learn how to use the environment quickly and easily. This principle has been derived from MAGPIE and ZSTEP (section 2.4) which both have a unified programming environment.

The novice will have to learn how to use an editor in order to write programs, and will learn the commands and environment early in their programming experience. It is therefore sensible to reduce the amount of information that the novice needs to know in order to use the tracing system, by basing that system on the editor. This will reduce the cognitive load on the novice when learning a language, allowing him/her to concentrate on learning to program rather than on how to use the different tools provided in the programming environment.

### *3.2.9 Demonstration Utility*

This principle is aimed at providing a means of presenting the user with a concrete example of programming techniques so that s/he may be aided in abstracting out general plans for programming. Jones (1984) has pointed out that programming success is dependent upon this ability to abstract out plans from examples.

The stepping facilities at the most detailed level should be able to be used to demonstrate different features of the programming language, i.e. backtracking, cut, function calls , and programming techniques such as recursion and search. This will allow the system to be used as a teaching tool, used either with an automated tutoring system, with lectures on programming, or just providing a pool of on-line dynamic demonstrations available to the user through a help system.

A demonstration facility should allow the novice programmer to associate concrete programs with the abstract algorithms they need to understand in order to become successful programmers.

### *3.2.10 Minimal Extraneous Symbols*

This principle aims at reducing the number of symbols that are used to flag the meaning of information that is presented to the user in the trace. This is another attempt to keep the display of the trace 'simple', by reducing the amount of information the novice needs to know before s/he can use the system.

Only the symbols used in the syntax of the language should be used in the trace-time code of the stepper display. Using other symbols to denote particular events in the trace causes several problems. Firstly it means that the novice has to learn the meaning of all the symbols before s/he can use the system, and usually has to learn the model of program execution that these symbols represent. This is not helpful as the aim of animated tracing tools is to facilitate the assimilation of a working model of program execution. Secondly, the symbols clutter the display when they are used distracting the user from concentrating on what is happening to the program.

The important features/events displayed by the animated tracing system should be highlighted using inverse video or colour, so that the text being highlighted is augmented in its importance with respect to the rest of the display. This concentrates the user's attention on the text that is highlighted rather than distracting it with extraneous symbols.

### *3.2.11 Non Proliferation of Views*

This principle states that the number of views a novice is given to show program execution should be kept to a minimum, and is another example of keeping the system 'simple'. Paxton and Turner (1984) have shown that novices prefer inflexible

tools, with few commands so that they can concentrate their attention on understanding one view of program execution instead of spreading it across several displays. Once the novice has built a working model of the execution of programs then other views could be made available, which the user can then understand in terms of the original view. This should allow a smooth progression in the complexity and number of views provided by the system as the user becomes more adept at using the programming language.

### *3.2.12 Detail/Speed Trade-off*

Novices need an inflexible detailed story of the 'virtual machine' allowing them to build a conceptual model of program execution, while experts want a flexible fast interface so that they can use their hypothesis about the action of the program in order to look for a particular piece of information and find it quickly. When designing an animated tracing system the designer should primarily follow this principle, because the system is aimed at novice programmers, but s/he should also design the system so that it can be integrated into a larger system of programming tools that can be used by the novice as s/he becomes more and more adept at programming. This approach should provide a smooth ascent for the novice into fast flexible tools.

### *3.2.13 Display Shape*

The shape and placement of the windows displaying the code should be determined by the shape of the code. For example, assembly code programs which are 'long and skinny' should be displayed with two windows side by side (horizontally), whereas Lisp programs tend to be rather fat, and so should be presented in windows which are on top of each other. This approach allows more code to be seen on screen with less wasted space, so providing a natural interface between the language and the user.

## 3.3 Summary

The design principles described above present precise guidelines for building an animated tracing tool which aims to provide novice programmers with a view of program execution which they can use as the basis for building a working model of how a programming language works. In order to demonstrate that these guidelines are both realistic and realisable it is necessary to build an animated tracer based on the principles specified above. This will also allow the design to be tested on novice user's which will indicate whether it reaches its aims of being simple to use, and communicates the desired run-time information.

In order to determine the limits of the above principled design in terms of its applicability to different types of programming language it is necessary to test the guidelines on several languages which are representative of both high/low level, and procedural/declarative languages.

Chapter 4 describes three 'canned' prototype tracers (APT-0) which are based on the design principles described above. They show an animated view of program execution for three programs one each for Prolog, Lisp and 6502 Assembler. The production of 'canned' prototypes provides an approach that will enable the above question to be answered. It also provides a means of building an animated view of program execution rapidly without the time consuming need for programming. Chapter 5 presents empirical studies which investigate whether novice's understand the display of program execution that APT-0 presents.

The prototypes being 'canned' only demonstrate the way the animated tracer works on one program for each language. To show that the approach presented in this chapter works for all the different features that are inherent in a complex programming language it is necessary to develop a real tracer that can cope with all these features. Chapter 6 provides implementation details for a real animated tracer for Prolog, and presents a sample scenario of the tracer in action.

Chapter 7 describes empirical studies which investigate the misconceptions that novice Prolog programmers hold concerning program execution, and the affect of APT on their ability to predict the action of Prolog programs at run-time. The second study aims to determine whether APT is successful in communicating information concerning program execution to novice programmers.

# CHAPTER 4

# THE PROTOTYPES OF APT

The following chapter discusses the features of APT-0, the prototypes of the animated program tracer. To determine the scope of the design principles described in the previous chapter, three prototype tracers were built for varying types of programming languages. The languages chosen Prolog, Lisp and 6502 Assembler range from high to low level, procedural to declarative, and interpreted to non-interpreted languages. The protoptypes of APT, from now on called APT-0, were built not as real systems but as 'canned' text, presented as an animation. This enabled each prototype to be built rapidly, changes made easily, and feedback could be gained quickly.

Following the description of the prototypes is an account of an evaluation of each prototype to determine the effectiveness of presenting an animated view of program execution to novice programmers. The aim of this evaluation was to produce some fast feedback enabling improvements to be made to the display before designing and building a real animated stepper for Prolog (see chapter 6).

## 4.1 Description of the Prototypes

First the general appearance of the interface, which is relevant to all three prototypes, will be presented. It should be noted that the Assembler prototype is slightly different to the Prolog and Lisp displays, and these differences will be

mentioned in the Assembler section below. This is followed by a short scenario for the Prolog, Lisp and 6502 Assembler stepper displays. As the different features of each prototype are presented you will notice comments in curly brackets i.e. {*Status line*}. These comments refer to the respective design principles described in the last chapter, which lie behind the feature being described.

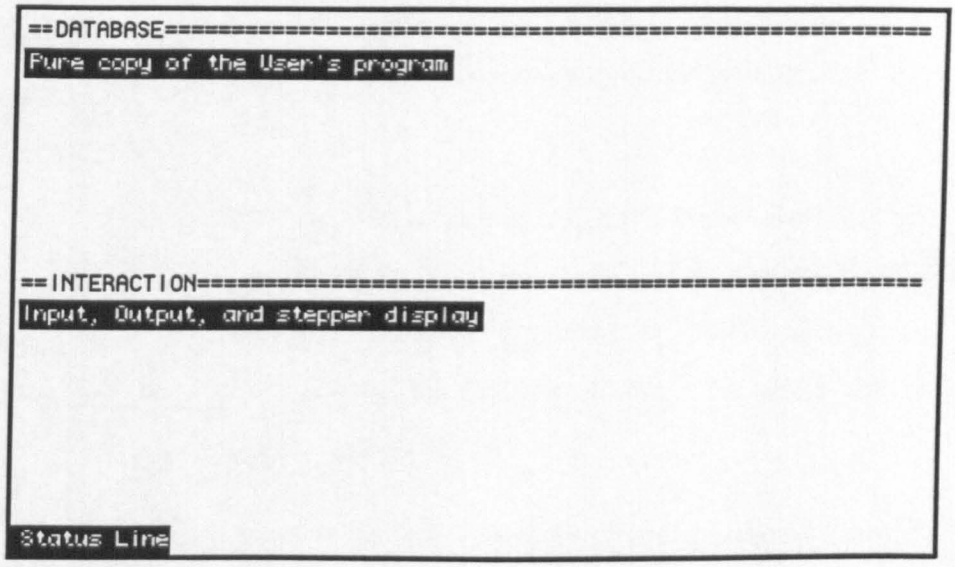



Figure. 4.1 The general appearance of the APT-0 display

Fig 4.1 shows the appearance of the APT-0 display, which was based on an experimental facility developed at the Open University for teaching Prolog to novices (Eisenstadt, Hasemer and Kriwaczek, 1984). The screen consists of two windows and a status line which can contain messages to the user in the form of text. The 'DATABASE' and 'INTERACTION' windows can be seen in the picture below their titles of the same name, with the status line beneath them at the bottom of the screen. The so called 'DATABASE' and 'INTERACTION' windows represent the kind of windows which support editing facilites that can be found in most word processors {*Environment uniformity*}. The top window is where the user would write his/her program, and it therefore contains a copy of the user's edit-time code. Also any side effects on the programming environment due to the program at run-time will dynamically update this window {*Side-effect visibility*}.

The 'INTERACTION' window represents a pseudo top-level. This means that the window should act in the same manner as the top-level of any interpreted language in that commands can be typed in and output printed out. The difference is due to the presence of the editing facilities which allow scrolling, editing and re-execution of old commands. Another difference in the use of this window is caused by the action of the animated tracer. When the tracer runs this window is used as a scratch pad or work space, where the execution of the of the user's program unfolds. This will be described later in detail.

In normal use, i.e. editing or running programs, the status line residing at the bottom of the screen remains blank. However, when the tracer runs this line is used to comment on the action of the program at every step. This informs the novice of what is happening in the execution of his/her program, and in fact tells an abbreviated story of the execution of the user's program.

The prototype stepper displays work by showing the novice user each line of his or her program being executed in the same order and in the same manner as the interpreter does (or compiler in the case of 6502 Assembler) {*True eval sequence*}. This animation of program execution takes place in the 'INTERACTION' window. In the case of the Prolog and the Lisp stepper this means that the display grows as new rules/functions are called, and is called in-place subroutine instantiation {*In-place*} (the animated Assembler display shows the side effects of the program on memory addresses {*Side-effect visibility*}). Figures 4.2 and 4.3, two frames from the Prolog prototype, show a goal call and the in-place instantiation of the rule that matches that call.

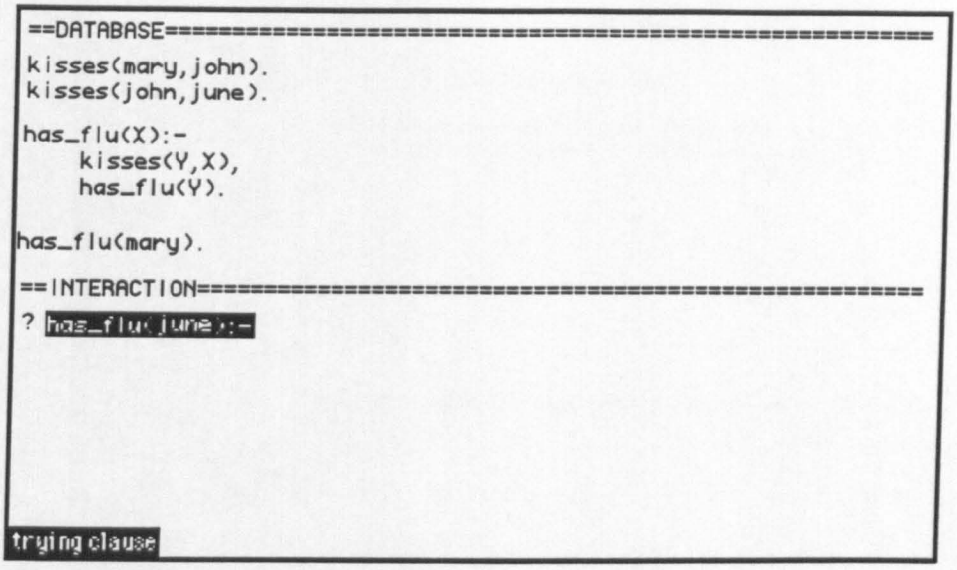



Figure. 4.2 A Prolog goal call in APT-0

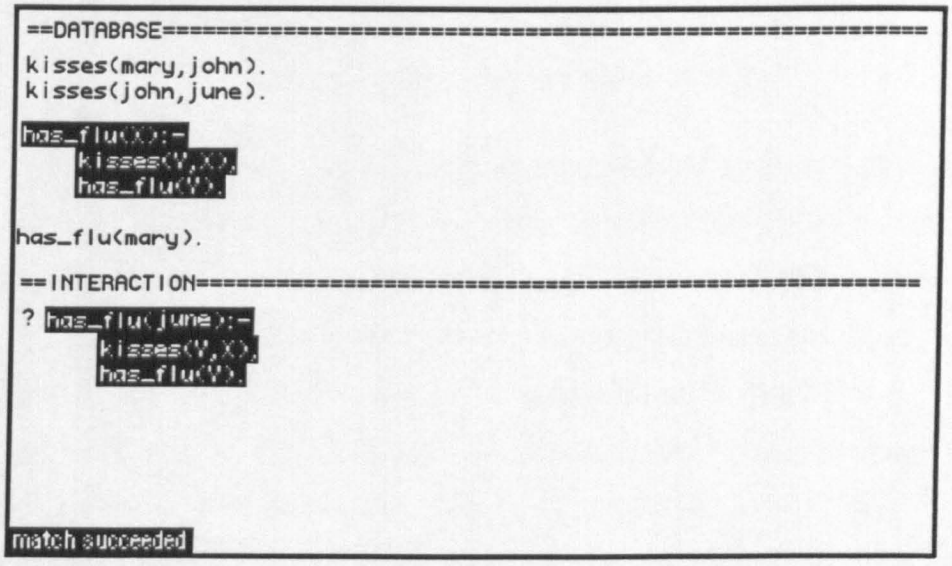



Figure. 4.3 In-place subroutine instantiation in APT-0

As well as the in-place subroutine instantiation, any variables that occur during the program are highlighted using inverse video and replaced by the value they hold. This enables the novice to see clearly the flow of control and the data flow. Figure 4.4 shows a frame from the Prolog prototype which demonstrates the highlighting of variables and their values. In the figure the variable 'Y' is highlighted in the 'DATABASE' window, while the value it holds, 'john', has replaced the 'Y' in the

'INTERACTION' window. Both the variable and the value are highlighted, and the binding of the variable is commented on by the status line.

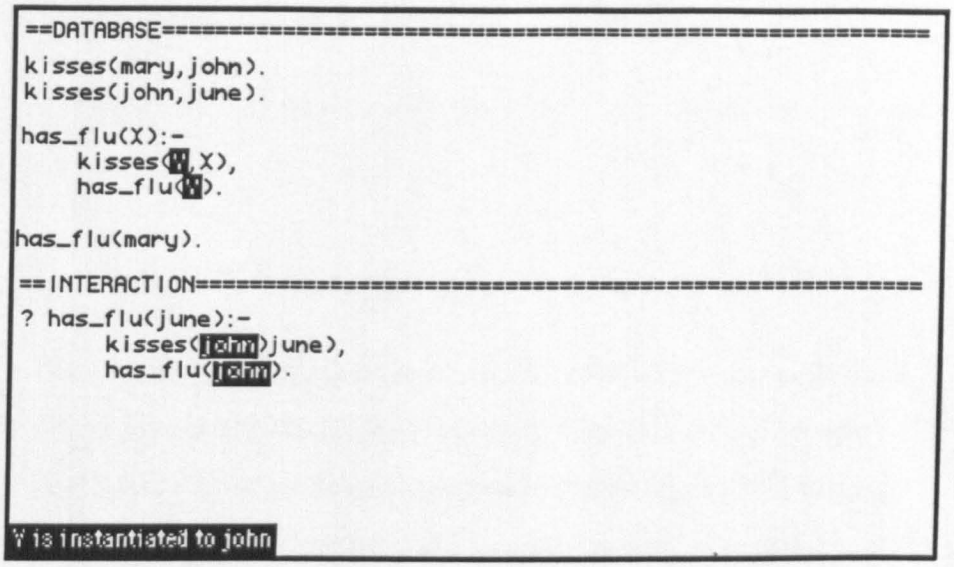



Figure 4.4 Highlighting of variables in APT-0

Both the edit-time and trace-time code are shown during the animation, the former in the 'DATABASE' window and the later in the 'INTERACTION' window. The correspondence between the edit-time and trace-time {*Edit = Trace*} code is conveyed to the novice by the means of inverse video highlighting {*Minimal symbols*}, which shows the important features at all times. In figure 4.4 the value 'john' in the trace-time code (DATABASE window) is associated to the variable 'Y' in the edit-time code (INTERACTION window) by the inverse video highlighting. The highlighter shows such things as the variable name, where its instantiation/binding is going to come from; matching of rules and facts; calling of functions; evaluation of s-expressions, replacing them with the value they return; and updating the database window. In addition to the information provided by the animation of program execution, there is the abbreviated story conveyed by the messages which appear on the status line. These messages can remind the novice of the buzzwords used in the explanations commonly found in programming manuals and teaching texts, and reinforce the animated trace displayed in the 'INTERACTION' window.

The user can move both forwards and backwards in each stepper display so that tricky pieces of code can be retraced without the user having to exit the tracer and start all over again {*Trace forwards/backwards*}.

The next three sections show examples of APT-0 for Prolog, Lisp, and 6502 Assembler.

### *4.1.1 Prolog*

The Prolog stepper display is the same as described above, consisting of the 'DATABASE' and 'INTERACTION' windows. The 'DATABASE' window contains a pure copy of the user's edit-time code {*Edit = Trace*}, which scrolls automatically during the animation to keep the relevant piece of program in view at all times. The stepper is called from the 'INTERACTION' window, which then shows the sequence of events that occur as the program is executed. The animation describes the order that goals and subgoals are matched against entries in the database {*True eval sequence*}, and the instantiation of variables due to unification {*Variable integration*}.

The story of variable instantiation given in the prototype is to instantiate all instances of the same variable at the same time. When a variable is being bound the variable name is highlighted in both the edit-time and trace-time code, allowing the user to see where the variable is in the original program. The variable name is then replaced on-screen with the value it holds.

Outstanding goals in the trace-time code are matched against procedure entries in the database one at a time, demonstrating how the interpreter searches through the database. The goal is highlighted in the trace-time code while each attempted match against clauses in the edit-time code (the database) is highlighted in turn. If the goal matches a rule then that rule is inserted into the display in the trace-time code so that its subgoals may be solved and its variables instantiated accordingly {*In-place; True*

*eval sequence*}. If the goal matches a fact then any variables that are uninstantiated are bound. If a goal fails backtracking is initiated. When this happens if the failed goal has any subgoals they are highlighted and any variables that where bound in these subgoals are uninstantiated. The failed subgoals are then removed from the display. The tracer then attempts to match the goal (the previous parent goal) against other database entries.

At all times throughout the animated trace the 'Status Line' provides brief messages informing the user in English what is happening at every step of the display.

Figures 4.5 to 4.8 show several snapshots of a prolog program being stepped by APT-0.

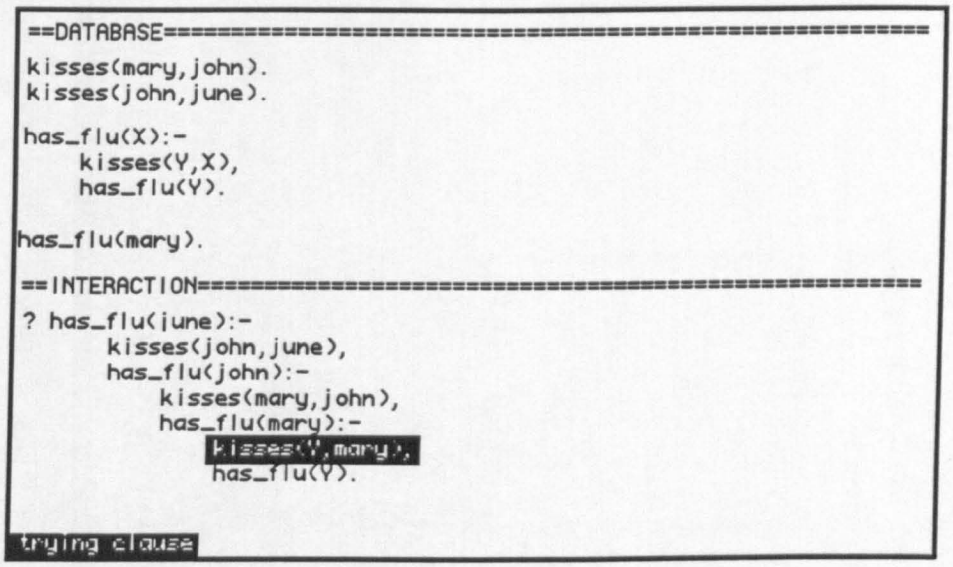



Figure 4.5 A series of screen snapshots from the Prolog APT-0

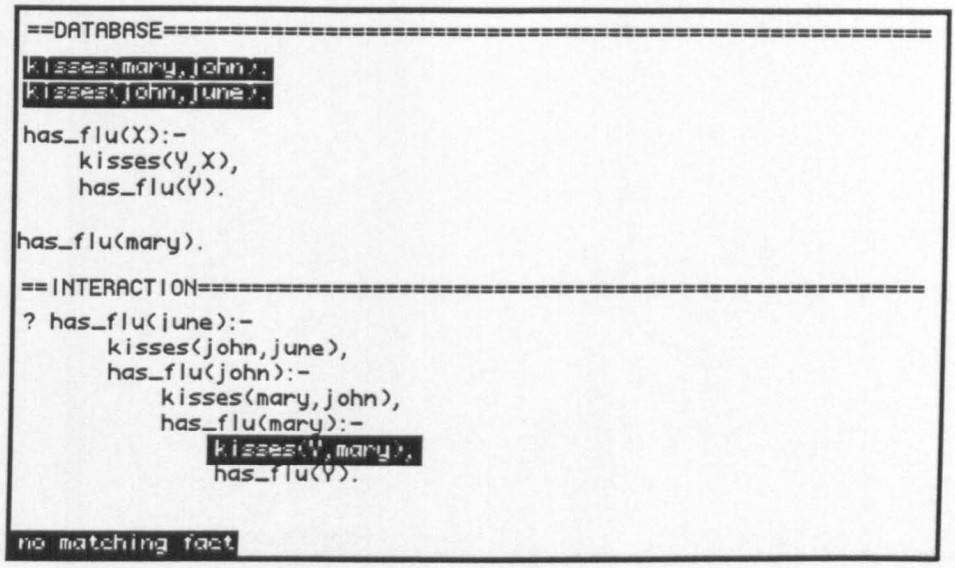



Figure 4.6

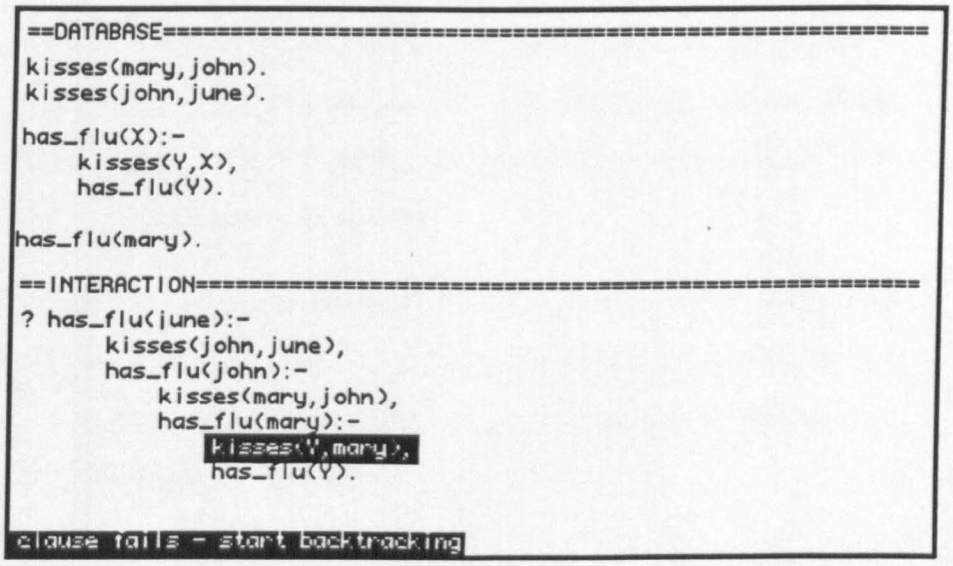



Figure 4.7

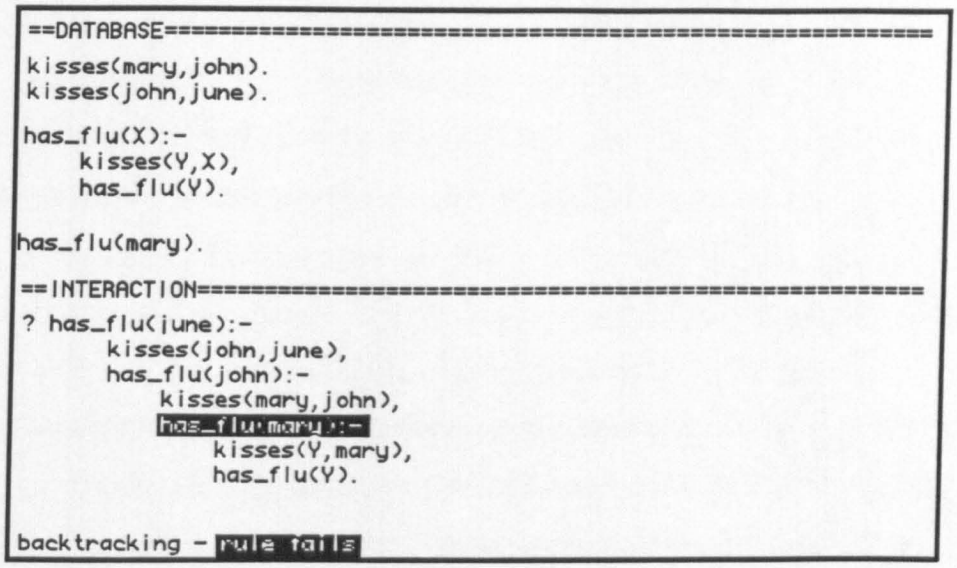



Figure 4.8

Fig 4.5 shows that the program has been partly stepped already beginning with the top-level goal 'has_flu(june)'. In this frame APT-0 is trying to match the clause 'kisses(Y,mary)', which is highlighted in inverse video. Notice that the status line message confirms this. {*Status line*}

Fig 4.6 shows that the clause 'kisses(Y,mary)', which is still highlighted, is not present in the database. This is done by highlighting those clauses that are similar to the clause that is being matched. The status line reports that 'no matching fact' can be found in the database. {*Edit = Trace*} {*Status line*}

Fig 4.7 displays that the highlighted clause 'kisses(Y,mary)' has failed and that backtracking is about to start. The status line message keeps the user informed as to what is happening. {*WYSIWHa*} {*Status line*}

Fig 4.8 tells the user that the highlighted (head of the) rule 'has-flu(mary)' has failed on backtracking. The status line confirms this. {*Status line*}

An important point to mention here is the detailed action of APT-0 when it backtracks or when it encounters the cut. When backtracking occurs a message

appears on the status line pointing this out to the user, 'rule fails - backtracking', and the previous goal is highlighted (the highlighting occurs in both the 'DATABASE' and 'INTERACTION' windows) {*Edit = Trace*}. Next the Prolog code that has failed and is to be deleted from the display is highlighted, with the following status line message, 'backtracking - remove failed code' {*Status line*}. The failed code is then removed from the screen. The goal that is to be retried is now highlighted, the status line displaying 'backtracking - retry <clause name>' {*Status line*}. If a cut is encountered on backtracking the same series of events as above happens, with the following additions. The status line will display the message, 'backtracking - cut encountered', with the cut symbol '!' highlighted in the 'trace-time' code {*Status line*}. The parent goal is highlighted, the status line showing, 'backtracking - cut encountered - parent goal fails' {*Status line*}. The failed code is then removed in the same sequence as before, but with the following status line message, 'backtracking - cut encountered - parent goal fails - remove failed code' {*WYSIWHa*} {*Status line*}.

### *4.1.2 Lisp*

The Lisp stepper is called from the 'INTERACTION' window, with the pure copy of the user's code is shown in the database window {*Edit = Trace*}. This window scrolls automatically to keep the relevant piece of code in the centre of the window. The 'INTERACTION' window shows the sequential evaluation of the edit-time code.

When functions are called from the trace-time code the definition of that function is highlighted in the edit-time code. This is followed by the function definition being inserted into the trace time code so that the evaluation of each part of that function can be animated. Each s-expression in the function is evaluated on-screen in the same sequence as the interpreter evaluates the function. S-expressions in the trace-time code are taken in turn and any variables they contain are highlighted along with the corresponding variable name in the edit-time code. The last value that the variable

held is also highlighted in the trace time code, as this is the value that the variable will now hold. The variable name in the trace-time code is replaced with the highlighted value. When all the variables have been bound the function is highlighted and evaluated. This highlighted function (in the trace-time code) is then replaced with the value that it has returned on evaluation. When the last s-expression has been evaluated the display winds back to the outer function and eventually arrives back at the starting point of the stepper call, finishing with a 'NIL' or 'T' printout (or any printout supplied by a user's function) {*Variable integration*}. This display of program execution makes clear exactly what is going on as the Lisp code is evaluated.

Notice that each function is displayed in the form the novice wrote it i.e. (defun foo (x) - - -) rather than in the underlying 'lambda' notation that some experts prefer {*Edit = Trace*}. The reason for this is that novices are more likely to recognise and understand the 'defun' format rather than the 'lambda' notation due to the isomorphic mapping between the edit-time and run-time code.

A status line is present on the bottom line of the display, which provides brief comments on the animated display at each step of the trace{*Status line*}. It is intended as a navigation aid, so that if the novice gets lost or confused a glance at the status line will inform him/her of the state of the stepper at that particular step e.g. 'X is instantiated to mary'.

The display can be run forwards, interpreting the user's code in the correct sequence, or backwards. The display may also be stopped at any point.

Figures 4.9 - 4.13 show part of a Lisp program being stepped through. Fig. 4.9 shows that the program has already been partly stepped through. The stepper has found a variable (in the first test on the 'cond'), which has been instantiated to " 'mary ". The parameter of the 'infect' function, " 'mary " has been highlighted to show where the value originated. The two x's in the database window have been highlighted for reference, showing the name of the variable that holds the value "

'mary " {*Minimal symbols*}. The status line brings the instantiation of the variable to the user's attention {*Status line*}.

```
==EDITOR=============================================================
(defun infect (x)
    (putprop x 'flu 'has)
    (cond ((eq (get x 'kisses) nil) nil)
          (t (infect (get x 'kisses) ))))

(putprop 'mary 'john 'kisses)
(putprop 'john 'june 'kisses)
(putprop 'mary 'flu 'has)

==INTERACTION========================================================
(infect 'mary)
(infect 'mary
    (putprop 'mary 'flu 'has)
    (cond ((eq (get 'mary 'kisses) nil) nil)
          (t (infect (get x 'kisses) ))))



Conditional :- x is instantiated to 'mary
```

Figure 4.9 A series of screen snapshots from the Lisp APT-0

```
==EDITOR=============================================================
(defun infect (x)
    (putprop x 'flu 'has)
    (cond ((eq (get x 'kisses) nil) nil)
          (t (infect (get x 'kisses) ))))

(putprop 'mary 'john 'kisses)
(putprop 'john 'june 'kisses)
(putprop 'mary 'flu 'has)

==INTERACTION========================================================
(infect 'mary)
(infect 'mary
    (putprop 'mary 'flu 'has)
    (cond ((eq (get 'mary 'kisses) nil) nil)
          (t (infect (get x 'kisses) ))))



Conditional :- evaluation
```

Figure 4.10

```
==EDITOR=============================================================
(defun infect (x)
    (putprop x 'flu 'has)
    (cond ((eq (get x 'kisses) nil) nil)
          (t (infect (get x 'kisses) ))))

'(putprop 'mary 'john 'kisses)
(putprop 'john 'june 'kisses)
(putprop 'mary 'flu 'has)

==INTERACTION========================================================
(infect 'mary)
(infect 'mary
    (putprop 'mary 'flu 'has)
    (cond ((eq '(get 'mary 'kisses) nil) nil)
          (t (infect (get x 'kisses) ))))



Conditional :- evaluation
```

Figure 4.11

```
==EDITOR=============================================================
(defun infect (x)
    (putprop x 'flu 'has)
    (cond ((eq (get x 'kisses) nil) nil)
          (t (infect (get x 'kisses) ))))

(putprop 'mary 'john 'kisses)
(putprop 'john 'june 'kisses)
(putprop 'mary 'flu 'has)

==INTERACTION========================================================
(infect 'mary)
(infect 'mary
    (putprop 'mary 'flu 'has)
    (cond ((eq '(get 'mary 'kisses) nil) nil)
          (t (infect (get x 'kisses) ))))



Conditional :- evaluation
```

Figure 4.12

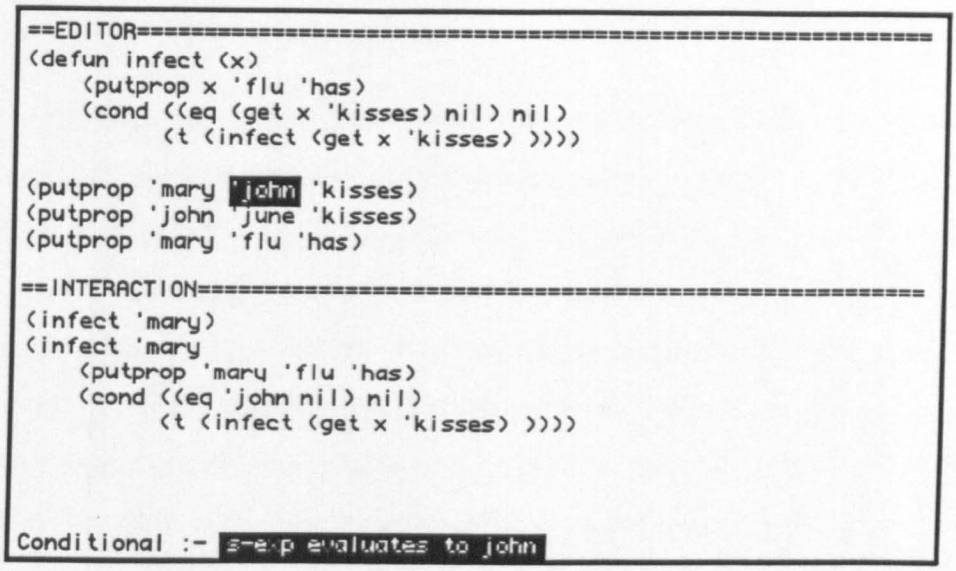



Figure 4.13

Fig. 4.10 moves on to focus on the "(get 'mary 'kisses)" s-expression, which is now highlighted. The status line mentions that it is to be evaluated.

Fig. 4.11 shows that the "(putprop 'mary 'john 'kisses)" has been additionally highlighted, indicating that the "(get 'mary 'kisses)" s-expression refers to this piece of code.

Fig. 4.12 shows that " 'john " is specifically the information required by the 'get' function, by only highlighting " 'john " in the "(putprop 'john 'mary 'kisses)".

Fig. 4.13 is the final frame in the sequence. The "(get 'mary 'kisses)" expression has been evaluated, and has returned the value " 'john ". This being the case the "(get 'mary 'kisses)" expression in the interaction window has been replaced with " 'john ". The status line brings this to the attention of the user, "Conditional :- s-exp evaluates to 'john".

### *4.1.3 6502 Assembly Language*

The Assembler display differs from the other two in several ways. The most obvious of these is that the two windows are split vertically instead of horizontally {*Display shape*}. The reason being that Assembler code tends to be long and skinny so is best represented like this. The left window is labelled with 'INTERACTION', and the right with 'DATABASE'. Again the windows represent text which may be edited as in a word processor {*Environment uniformity*}. The stepper is split into two passes, as is 6502 Assembler when it is run {*True eval sequence*}. The first part shows the user's Assembler code in the left window, and a blank right window. The code is stepped through looking for constants and their associated values. When one is found it is transferred to the right window for later reference. From here on every time a constant is found in the 'trace-time code it is replaced with its associated value. This is shown by relating the constant in the 'INTERACTION' window to the constant and value in the 'DATABASE' window, using inverse video as a highlighter.

The second part of the display executes the assembler code. The right hand window is made blank, and initialised with boxes representing the accumulator, index registers and flag register. Each instruction is executed showing how it affects the flag register, or how the flag register affects it, and the accumulator and index registers are updated accordingly {*True eval sequence*; *Variable integration*}. When an instruction that deals with a memory address or its contents is executed, that particular address and its contents are displayed in the 'DATABASE' window, together with the address and contents on either side of it . The memory contents are updated dynamically according to the instructions given.

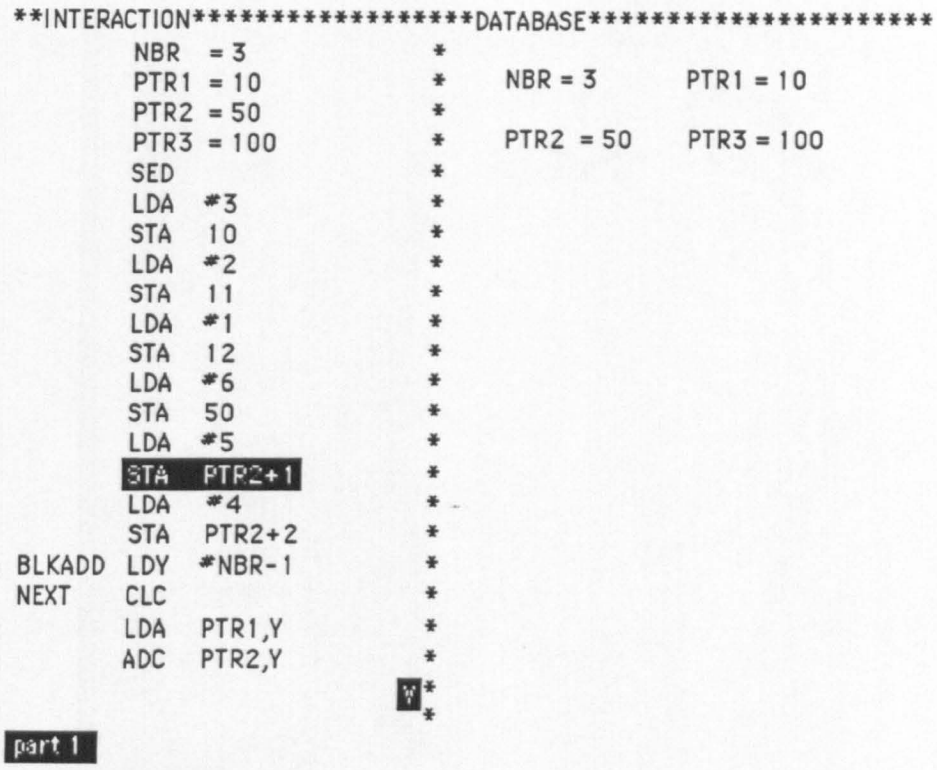



Figure 4.14 A series of screen snapshots from the Assembler APT-0

```
**INTERACTION******************DATABASE**********************
        NBR  = 3              *
        PTR1 = 10             *   NBR = 3       PTR1 = 10
        PTR2 = 50             *
        PTR3 = 100            *   PTR2 = 50     PTR3 = 100
        SED                   *
        LDA  #3               *
        STA  10               *
        LDA  #2               *
        STA  11               *
        LDA  #1               *
        STA  12               *
        LDA  #6               *
        STA  50               *
        LDA  #5               *
        STA  PTR2+1           *
        LDA  #4               *
        STA  PTR2+2           *
BLKADD  LDY  #NBR-1           *
NEXT    CLC                   *
        LDA  PTR1,Y           *
        ADC  PTR2,Y           *
                              *
                              *

part 1 : mnemonic
```

Figure 4.15

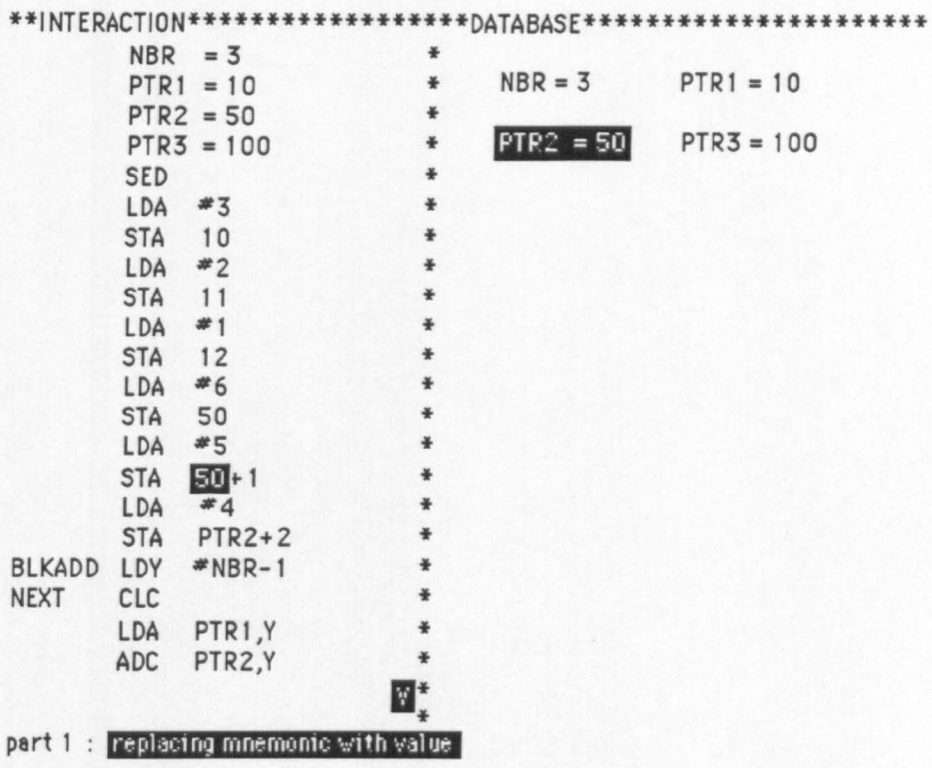

**INTERACTION**********DATABASE**********
NBR = 3
PTR1 = 10
PTR2 = 50
PTR3 = 100
SED
LDA #3
STA 10
LDA #2
STA 11
LDA #1
STA 12
LDA #6
STA 50
LDA #5
STA 50+1
LDA #4
STA PTR2+2
BLKADD LDY #NBR-1
NEXT CLC
LDA PTR1,Y
ADC PTR2,Y
NBR = 3
PTR1 = 10
PTR2 = 50
PTR3 = 100
part 1 : replacing mnemonic with value

Figure 4.16

```
**INTERACTION******************DATABASE**********************
          NBR  = 3                *
          PTR1 = 10               *    NBR = 3       PTR1 = 10
          PTR2 = 50               *
          PTR3 = 100              *    PTR2 = 50     PTR3 = 100
          SED                     *
          LDA  #3                 *
          STA  10                 *
          LDA  #2                 *
          STA  11                 *
          LDA  #1                 *
          STA  12                 *
          LDA  #6                 *
          STA  50                 *
          LDA  #5                 *
          STA  50+1               *
          LDA  #4                 *
          STA  PTR2+2             *
BLKADD    LDY  #NBR-1             *
NEXT      CLC                     *
          LDA  PTR1,Y             *
          ADC  PTR2,Y             *
                                  *
                                  *
part 1 : evaluation
```

Figure 4.17

```
**INTERACTION*****************DATABASE**********************
         NBR  = 3              *
         PTR1 = 10             *    NBR = 3       PTR1 = 10
         PTR2 = 50             *
         PTR3 = 100            *    PTR2 = 50     PTR3 = 100
         SED                   *
         LDA  #3               *
         STA  10               *
         LDA  #2               *
         STA  11               *
         LDA  #1               *
         STA  12               *
         LDA  #6               *
         STA  50               *
         LDA  #5               *
         STA  51               *
         LDA  #4               *
         STA  PTR2+2           *
BLKADD   LDY  #NBR-1           *
NEXT     CLC                   *
         LDA  PTR1,Y           *
         ADC  PTR2,Y           *
                               *
                               *

part 1 : evaluates to 51
```

Figure 4.18

```
**INTERACTION*****************DATABASE**********************
        SED                  *
        LDA  #3              *   |A| 3 |    |X| |    |Y| |
        STA  10              *
        LDA  #2              *   | 9|   |
        STA  11              *   |10| 3 |
        LDA  #1              *   |11|   |
        STA  12              *
        LDA  #6              *
        STA  50              *
        LDA  #5              *
        STA  51              *
        LDA  #4              *
        STA  52              *
BLKADD  LDY  #2              *
NEXT    CLC                  *
        LDA  10,Y            *
        ADC  50,Y            *
        STA  100,Y           *
        DEY                  *
        BPL  NEXT            *   N V   B D I Z C
                             *   | 0 | | | 1 | | 0 | | | |
part 2
```

Figure 4.19

```
**INTERACTION*****************DATABASE**********************
        SED                  *
        LDA  #3              *   |A| 2 |    |X| |    |Y| |
        STA  10              *
        LDA  #2              *   | 9|   |
        STA  11              *   |10| 3 |
        LDA  #1              *   |11|   |
        STA  12              *
        LDA  #6              *
        STA  50              *
        LDA  #5              *
        STA  51              *
        LDA  #4              *
        STA  52              *
BLKADD  LDY  #2              *
NEXT    CLC                  *
        LDA  10,Y            *
        ADC  50,Y            *
        STA  100,Y           *
        DEY                  *
        BPL  NEXT            *   N V   B D I Z C
                             *   | 0 | | | 1 | | 0 | | 0 |
part 2 : execution
```

Figure 4.20

Figures 4.14 to 4.20 show part of a 6502 Assembly program being stepped.

Fig 4.14 shows a frame from 'part 1' of the APT-0 display. The highlighted instruction 'STA PTR2+1' is being focused on by the stepper. The status line informs the user that this is the first part of the display , and the highlighted 'V' denotes that there is more of the program out of sight of the display (this 'V' would be replaced if a more sophisticated graphics terminal were being used, however there are not any satisfactory ways of displaying this point on a VT100 terminal). {*Status line*}

Fig 4.15 has picked out the constant 'PTR2' which has been highlighted in both the 'edit-time' (the 'DATABASE' window) and the 'trace-time' (the 'INTERACTION' window) code. The status line gives the message, 'part 1 - mnemonic' to inform the user of what 'PTR2' is (see section 7, the term mnemonic was found to be misleading and will be changed to the term 'constant' in future developments) {*Edit = Trace*; *Status line*}.

Fig 4.16 is the next frame in the sequence, and shows that the constant 'PTR2' has been replaced with its value '50'. The value '50' is highlighted in both windows to show where the value has come from. The status line tells the user that this frame is showing the replacement of a mnemonic with its value {*Edit = Trace*; *Status line*}.

Fig 4.17 shows the evaluation of the arguments to 'STA' . The arguments are highlighted, and the status line confirms tht they are about to be evaluated {*Status line*}

Fig 4.18 displays what the arguments to 'STA' evaluate to. The result of the evaluation '51' replaces the two arguments, and the status line also tells the user the result of the evaluation {*Status line*}

Fig 4.19 moves on to the second part of the APT-0 display. The frame shows that the display is focusing on the instruction 'LDA 2', which is highlighted. The

parts of the database that are associated with this instruction are highlighted also. These include the accumulator register and the negative, zero and carry flags of the flag register. The status line indicates that this is 'part 2' of the display {*Edit = Trace*; *Side effect visibility*; *Status line*}.

Fig 4.20 shows the 'LDA 2' instruction, which is still highlighted, being executed. The effect on the accumulator, i.e. inserting the value '2', is displayed by inserting and highlighting the figure '2' in the accumulator in the 'DATABASE' window. The changes in the values of the appropriate flag registers are also displayed with highlighting. The status line reminds the user that the highlighted instruction has been executed {*Edit = Trace*; *Side effect visibility*; *Status line*}.

## 4.2 Summary

This chapter has presented three prototype tracing tools, one each for Prolog, Lisp and 6502 Assembler. Each prototype embodies the design princples described in chapter 3, and displays a detailed, yet simple, view of program execution which provides explicit information about the events that happen to a program at run-time.

As discussed in chapter 2, the majority of the problems that novice programmers encounter concern dynamic concepts such as variable binding; function/goal invocation; flow of control; and recursion. Chapter two also shows that in order for novices to become successful programmers they need to be able to abstract out higher level programming plans which can be used as templates for future programming tasks, and how to combine specific language commands to form meaningful units of program code. The animated view of program execution that the prototypes present aim to answer these problems.

The principled APT-0 display allows the user to see what happens to a program when it is executed, and provides novices with the following insight into the how's and why's of programs at run-time.

The visual format of the trace-time display is based on the text of the edit-time. The trace-time code shows the sequence in which the program is executed by unfolding the run-time action of the program as execution happens. The simultaneous view of the edit and trace-time code allows the user to recognise the code used in the trace, and to associate the dynamic action of the trace-time code to the static edit-time code. This association is augmented by the use of inverse-video highlighting which picks out the important features of the display in both the trace and edit-time code. As each new function/goal is called, it is inserted into the trace-time display at the place it is invoked. The binding of variables is shown integrated in the trace-time code, and may help novices see where and how variables got their values, and how these values effect the execution of the rest of the program. At various points in the display the status line comments on what is happening in the trace. These brief messages help the user understand what is happening, by giving a commentary in terms of the language specific phrases which are used in all programming manuals to describe the execution of programs.

At a more global level the APT-0 prototypes provide a simple environment in which novice programmers can watch the execution of programs. The display shows only one view of the run-time action of a program, in which all aspects of execution which are relevant to novices are shown. The manner in which the execution is presented is consistent throughout the display. Specifically this means that each run-time action of the program has the following three display features associated with it: (1) some run-time action happens; (2) inverse video highlighting picks out this action, and relates it to the edit-time code; (3) the status line comments on the run-time action. Lastly the mechanism by which the user controls the tracer is simple. It consists of three commands which move the display forward a step; backwards a step; or stops the trace.

The simple and consistent approach allows the novice to concentrate on watching and assimilating the information the animated trace is providing about program execution, rather than worrying about how to control the tracer.

The fact that prototypes have been developed for such widely differing languages as Prolog, Lisp and 6502 Assembler demonstrates that the design principles underlying the animated display can be generalized to many programming language types. This means that it is possible to develop common programming and debugging tools enabling programmers to switch between languages without having to switch between programming environments with their different sets of commands. This should help reduce the cognitive burden placed both on novices learning new languages, and experienced programmers switching between languages. This approach to the design of programming environments allows the user to interact more efficiently with a programming environment.

# CHAPTER 5
# EVALUATION OF THE PROTOTYPES

The prototype APT-0 versions of the steppers show that the philosophy of dynamic tracing is possible for a range of different types of language. However, in order to see the effect of the prototypes on novice programmers and determine how the design features affect novices' ability to understand the stepped presentation of a program, protocols were taken of two novices viewing each prototype. In addition to the verbal protocols, timings were taken to determine how long the novice spent viewing each frame of the animation. It must be stressed that this study is not an experiment in the psychological sense, and was not meant to produce a detailed analysis of the different features of APT-0, nor of the design principles used in its design. In the rest of this chapter I shall continue to refer to these empirical studies for the sake of brevity, but the reader should remember that no hypothesis testing is involved. The aim of the data collection was to highlight the areas of the APT-0 prototypes that caused subjects problems, so that these areas could be improved for the implementation of APT-1.

## 5.1 Experimental Method

Three subjects, people with little or no programming experience in the particular programming language (apart from one who had a fair knowledge of Assembler programming), were given an instruction sheet consisting of a few lines describing the APT-0 stepper and a listing of the program to be stepped. From here on the subjects will be refered to as MS, IP, and AE.

A tape recording of each subject's detailed interactions and comments was taken so as to be able to analyse the protocols at a later date. The subjects were asked to describe the purpose of the program, to determine whether they had an understanding of the program to start with. They were then asked to step the program and describe verbally what was happening at each frame of the display. As well as the verbal protocols a dribble/log file was taken of the subject's movements through the stepper display, i.e. forwards and backwards movements, together with the time they spent on each frame.

The collection of timing data is a useful indication of how subjects interact with a display based computer interface. Timing information shows how long subjects have spent viewing specific parts of the system. By studying the timing data and determining which parts of the system subjects spent a long time viewing, and which parts they viewed briefly, some indication of how well subjects have understood each part of the system can be built up.

However this data only gives a coarse analysis of how each subject is interacting with the computer interface. Also the time subjects spend viewing sections of the interface only gives a relative measure of the ease with which they have understood the information that the display contains. This measure is relative because if a subject spends a long time viewing a part of the interface it does not mean that the display is hard or very hard to understand, but that it indicates that this section is more difficult to understand than another section which the subject spent less time viewing.

The protocol data supports the coarse analysis of the timing data by providing a more detailed picture of what the subject is thinking and doing as s/he interacts with the computer. This provides a mechanism which prevents the experimenter from drawing the wrong conclusions from the timing data. But wheras protocol analysis takes a long time to carry out, and the criteria used to do so are difficult to agree upon, the advantage of the timing anlaysis is that it provides a method of analysing data quickly with easily agreed criteria to give a global view of the subject's interaction.

## 5.2 Results

The results from this experiment are subdivided into two parts. The first part attempts to quantify the affects of APT-0, by looking at the time the subject spent viewing frames depicting similar features. For example the instantiation of variables, the evaluation of an expression, or the matching of a goal. The second part looks at key comments from the subject's protocols. Finally I will comment on how these results affect the design of APT-0 and improvements that can be made. Full transcriptions of the protocols and detailed presentation of the results can be seen in Appendix J.

The timing studies consist of three measures of the subjects' viewing time on APT-0 frames. For each of the three prototypes the frames that make up the animation are classified into categories presenting similar language features. Each subject is considered separately, and for each category of stepper frames a mean and median viewing time has been calculated. A graph is also presented which shows the cumulative percentage of frames which a subject viewed against time. For example in figure 5.3 60% of the frames viewed by AE were viewed in 4.3 seconds or under. This representation allows us to see a global view of how long subjects spent viewing frames from the stepper. More importantly it provides a basis from which to take a midpoint time for the analysis of the time spent viewing parts of the stepper.

The timing studies are presented in the following way. Each language is taken separately, and its frame categories are described. Each category represents the actions carried out in APT-0 when presenting the execution of a particular language's features, for example, matching clause heads; variable instantiation, and procedure calls. This enables an evaluation of how APT-0 fared in presenting different aspects of the programming languages. Each subjects' mean and median times for each frame category is presented in chart form, along with a cumulative graph showing the percentage of frames viewed against time. Following the timing studies for the

subjects in each APT-0 display are the verbal protocols taken from the subjects as they were viewing the displays.

Finally an analysis of this information refers back to the categories, and the APT-0 stepper.

The figures for the mean and median are rounded up to the first decimal place, and are given in seconds. In the cumulative data, the percentage of frames viewed at or under various times are also rounded up to the first decimal place, and do not therefore always add up to one hundred. The times here are also given in seconds.

A marker level has been taken from the cumulative frame graphs (for each subject) to compare against the mean and median times of each categorised group. This marker figure has been taken (arbitrarily) as the range of times the subject spent viewing 40% to 60% of the frames in the APT-0 display. The reason for this is to give a range of times that may be compared to the mean and median scores calculated for each subject. The marker level is felt to represent the time taken by a subject in viewing a frame that has caused at most only minor problems, the time being spent on reading the content of that frame and assimilating the information. It was then assumed that times compared with this marker level could be taken as a measure of the understanding that subjects had of the frames representing a particular category.

Any comments (printed in italics and contained in curly brackets) refer to the design principles described in chapter 3, and mean that this principle underlies the feature being presented.

### Rubric for Interpretation of Results

To help interpret the data presented per category, for each subject, I will make some comments here on what particular types of result mean. Comments are presented both for 'median' and 'mean' results, and have a mnemonic name (in

capitals) for easy reference. These comments are referred back to when the subject's category data is presented in the charts below.

Comments on median time

"OK': Where the median time is roughly the same as the marker range it can be said that the subject understood the frames that were presented in this category.

"CONFUSED': If the median time is higher than the marker range then it is likely that the information contained in the frames from this category have not been understood by, or have confused, the subject.

"EASY': When the median is lower than the marker range it can be assumed that the frames in that category have been easy for the subject to understand.

Comments on mean time

"CONFUSED' or 'OK': Where the mean time is higher than the marker range, it can have two meanings. If the median time is also higher than the marker then the frames in this category have caused the subject problems in understanding and assimilating their content. However if the median time is similar to, or lower than that of the marker range, then these problems are confined to only a few of the frames in the category.

"CONFUSED' or 'OK': If the mean time is similar to the marker range, there are again two possible meanings. The first is where the median time is similar to, or lower than the marker. In this case the comment is the same as for 'OK' - median time, i.e that the subject has understood the information contained in the frames of this category. On the other hand where the median time is higher than the marker range it can be seen that the mean time is at the higher end of the marker range. The interpretation of this should be the same as for 'CONFUSED' - median time, i.e. that the subject has had problems in understanding the frames in this category.

"EASY': There is only one case where the mean time is lower than the marker range. This is when the median time is also lower than the marker range. The indication from this is the same as for 'EASY' - median time, i.e. that the subject has encountered no problems.

### *5.2.1 Prolog Prototype*

This section presents the results from the study of the Prolog prototype tracer.

Categories

'NO MESSAGE': Frames that have no message on the status line.

'HIGHLIGHT GOAL': Attempting to satisfy rules/clauses, and goals that are satisfied. This entails highlighting the goals or part of goals in both the 'edit-time' and 'trace-time' code. {*Edit = Trace*; *Status line*}

'MATCHING CLAUSE': Selecting a rule/clause to try and satisfy a goal/subgoal. The rule/clause to be tried is highlighted in the 'edit-time' code, and instantiated in the 'trace-time' code. {*Edit = Trace*; *In-place*; *Status line*}

'VARIABLE': Encountering and instantiating variables. Each instance of that variable is highlighted in both the 'edit-time' and 'trace-time' code. The value is highlighted, then it is substituted for the variable. {*Edit = Trace*; *Variable integration*; *Status line*}

'BACKTRACKING': The head of the goal is highlighted in the 'trace-time' code, the failed 'trace-time' code is removed from the screen, and and attempt is made to resatisfy the goal. {*Status line*}

| Category | Mean time secs | Comment | Median time secs | Comment |
|---|---|---|---|---|
| NO MESSAGE | 28.1 | EASY | 4.1 | EASY |
| HIGHLIGHT GOAL | 9.0 | EASY | 4.0 | EASY |
| MATCHING CLAUSE | 4.3 | OK | 2.6 | OK |
| VARIABLE | 4.0 | OK | 2.1 | OK |
| BACKTRACKING | 4.0 | EASY | 4.5 | EASY |

Figure 5.1 Prolog category data for AE

| Category | Mean time secs | Comment | Median time secs | Comment |
|---|---|---|---|---|
| NO MESSAGE | 13.4 | EASY | 15.4 | OK |
| HIGHLIGHT GOAL | 19.4 | OK | 17.5 | OK |
| MATCHING CLAUSE | 18.9 | OK | 19.6 | OK |
| VARIABLE | 22.0 | OK | 19.8 | OK |
| BACKTRACKING | 13.1 | EASY | 9.9 | EASY |

Figure 5.2 Prolog category data for IP

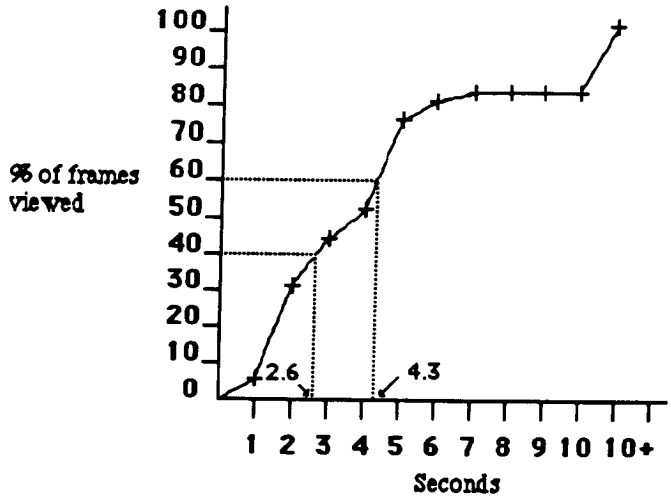



Figure 5.3 Cumulative graph for Prolog AE

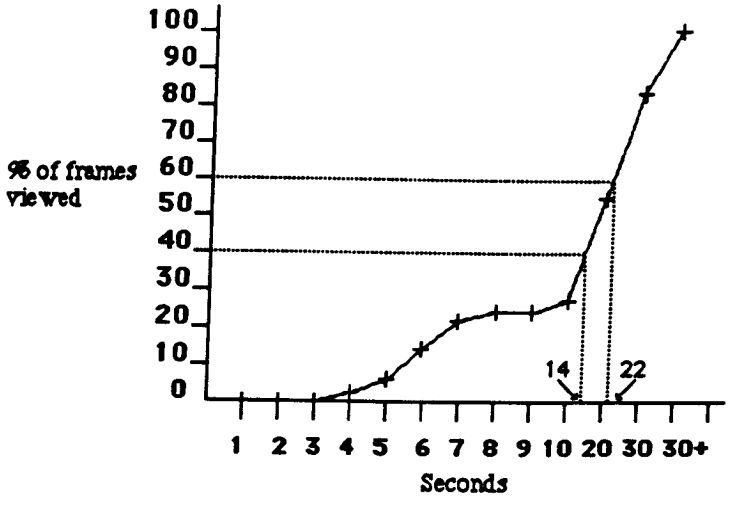



Figure 5.4 Cumulative graph for Prolog IP

Comments from Protocols

AE: (see appendix J for full listing of protocol)

AE was confused at one point in the display about whether a fact had been matched or if it still had to be matched,

*AE: Ehm right so now we have has flu John and presumably it's going to*

*no we've got kisses John June so it's going to try and match that, lets go back*

*it's already matched that*

*EX: It's trying it on for size, if you look at the comments*

*AE: Oh right so that's just saying that the clause has succeeded.*

this was clarified at once by a glance at the status line. After this point AE had no problems with the display of the program, and predicted the behaviour of the program correctly.

After the experiment AE commented that because he had some understanding of backtracking he understood the way it was presented in the display. However if he had not come across it before he might have been confused.

**IP:**

IP seemed to have some understanding of the Prolog program when he read it through. He understood the how the program should be read, and that 'June' would get flu via the chain of kissing, but he did not seem to be able to relate the two together,

*IP: Well it looks to me as though it's a bit peculiar. Er it looks like.. is it Mary kisses John, and John kisses June. Is that right.. yes.. and then it says if X, sorry, X has flu if X kisses Y and Y has flu. And then it says Mary has flu. So presumably you're going to find out em then that would say that John has flu 'cause he kisses Mary.*

*EX: Yes carry on, I want you tell me.*

*IP: Ah right. Because John's been kissing Mary, and June kisses John therefore June will get flu as well. Plus anyone else not mentioned here.*

When reading this program IP did not notice the presence of recursion, which he said later that he does not fully understand.

The first problem IP has with the display is that he is not sure of the relevance of the word 'INTERACTION' which is the title of the lower window. He does

however immediately recognize the 'edit-time' version of the code in the 'DATABASE' window,

*IP: So this is the same that's on here, so I want to go forwards now, as I've already explained what's on here.*

IP's predictions of what happens in Prolog are in larger jumps than the stepper display shows,

*IP: Now it's looking for the Y and having matched that I would say it will instantiate I suppose has flu Y will be has flu John*

*Yes I was right*

*Did I miss something out. Yes a clause it seems kisses John June has flu John*

*Matching head, did I miss something. I'm going to go backwards.*

IP does not understand the action of backtracking and is confused by the removal of failed code upon backtracking,

*IP: Oh no it doesn't so then it's going to look. Having done that it will go to the next rule which is has flu Y which will look to see whether Mary has flu and I happen to know that it doesn't have flu... in another part of this rule*

*so I've missed out describing the start of backtracking by saying what it's going to do next. But I still think I'm right*

*No I'm wrong, ha fails so it starts backtracking like it said*

*... which means that it goes back to where it started from*

*has flu Mary fails. Oh I see it fails on that count but because it's backtracking I imagine it's going to try on the second part of the rule which is has flu Y*

*Remove failed code, don't know what that means*

*Right*

*and then it does what I said it would do, it would do retry has flu Mary. Yes.*

The status line was found to be a great help, the protocol shows that IP read this line continually as a commentary on the action of the program.

IP's comments after the experiment brought up some interesting points. He found that when he looked at each new stepper frame, he was focusing on the word 'INTERACTION' rather than on what was going on in the window below. As with

AE's comments IP found the backtracking sequence confusing, particularly the part where the failed code is removed from the screen. On the plus side he very much liked the animated highlighting of the action of the program, as it clearly showed what was happening. IP found the status line helpful because it reminded him of what was going on in each frame. He thought that the level of detail presented in the APT-0 display was about right. Any less and he would not have been able to follow the display.

Improvements

In the APT-0 Prolog display only one category received a high viewing time and that was the backtracking category. In addition to this the protocols show that both subjects had problems with the backtracking sequence. This seems strange at first sight because figures 5.1 and 5.2 show that both AE and IP found backtracking easy which is contrary to the protocol data. The explanation is that the analysis of the timing data which appears in figures 5.1 and 5.2 present the mean and median values of the time the subjects spent viewing the different categories of stepped frames. However because subjects could move forwards and backwards through the stepped display they could view each frame more than once. If this happened the analysis would not see it as one frame viewed for a long time but several frames viewed for a short time. This will keep the mean and median time low which can be misleading. A more extensive analysis of the data taking into account frames that had been viewed more than once would provide more detail concerning the subjects usage of the stepper. This illustrates how verbal protocol and timing data can be used together in a supportive manner.

The main problem with backtracking seems to be the presentation of the removal of the failed code. I have as yet no ideas on how to solve this on an ordinary terminal screen other than an improved status line message. It would be possible to have a branching representation on a high resolution graphics display, so that the failed code remains in place but the 'trace-time' display branches off at the backtrack point. This

method could also be used when a 'cut' was encountered. However this would require either the addition of another view of program execution, i.e. a graph representation, or a terminal of very high resolution and screen size enabling branching of text. The latter is not possible with present day terminals and the former goes against one of the principles presented in chapter 3. This leaves the solution of the improved status line message.

Both categories 'NO MESSAGE' and 'HIGHLIGHT GOAL', which are those frames that had no status line message and frames dealing with the satisfaction of goals respectively, had some frames which caused long viewing times. IP also spent some time viewing these categories (although his viewing times did fall within the marker range), and AE's protocol showed some confusion with the matching of facts. This again suggests the addition of a status line message to improve the communication of the action of the program at run time.

Improvements can be made to those frames with no status line message, by giving a message indicating exactly what the stepper is doing; even if it is only telling the subject that APT-0 is focusing on the next item in the program. The problem with category 'HIGHLIGHT GOAL' frames may be solved by giving the user a status line message which interprets the action of moving 'edit-time' code into the 'trace-time' code in a more explicit manner. It is also possible that the long viewing times here may be due to the unusual practice of taking part of the 'edit-time' code and splicing it into the 'trace-time' code, in which case this would cease to be a problem after the display had been used a couple of times.

### *5.2.2 Lisp Prototype*

This section presents the results from the Lisp prototype tracer.

Categories

'NO MESSAGE': Frames that have no message on the status line.

'PROCEDURE': Procedure calls. The procedure call is highlighted in the 'trace-time' code, and the procedure is highlighted in the 'edit-time' code. The procedure is then instantiated in the 'trace-time' code. {*Edit = Trace*; *In-place*}

'VARIABLE': Encountering and instantiating variables. That particular instance of the variable is highlighted in both the 'edit-time' and 'trace-time' code. The value is highlighted, then it is substituted for the variable. {*Edit = Trace*; *Variable integration*; *Status line*}

'COND CLAUSE': Conditional clauses. The appropriate part of the conditional clause that is being worked upon is highlighted in both the 'edit-time' and 'trace-time' code. {*Edit = Trace*; *Status line*}

'EVAL S-EXP': Evaluation of S-expressions. The S-expression to be evaluated is highlighted in both the 'edit-time' and 'trace-time' code, and then the value the S-expression returns is substituted for it in the 'trace-time' code. {*Edit = Trace*; *In-place*}

'SIDE-EFFECT': Addition to the database. The information to be added to the database is highlighted in the 'trace-time' code, and then inserted into the database. {*Edit = Trace*; *Side-effect visibility*}

| **Category** | **Mean time** secs | **Comment** | **Median time** secs | **Comment** |
|---|---|---|---|---|
| NO MESSAGE | 17.5 | OK | 2.1 | OK |
| PROCEDURE | 9.9 | CONFUSED | 11.3 | CONFUSED |
| VARIABLE | 3.5 | OK | 2.0 | EASY |
| COND CLAUSE | 4.9 | OK | 2.4 | OK |
| EVAL S-EXP | 4.7 | OK | 3.0 | OK |
| SIDE-EFFECT | 5.1 | CONFUSED | 3.4 | CONFUSED |

Figure 5.5 Lisp category data for AE

| Category | Mean time secs | Comment | Median time secs | Comment |
|---|---|---|---|---|
| NO MESSAGE | 14.4 | OK | 15.5 | CONFUSED |
| PROCEDURE | 37.0 | CONFUSED | 24.1 | CONFUSED |
| VARIABLE | 16.6 | OK | 8.4 | EASY |
| COND CLAUSE | 14.5 | OK | 10.5 | OK |
| EVAL S-EXP | 17.5 | OK | 11.3 | OK |
| SIDE-EFFECT | 14.4 | OK | 10.8 | OK |

Figure 5.6 Lisp category data for IP

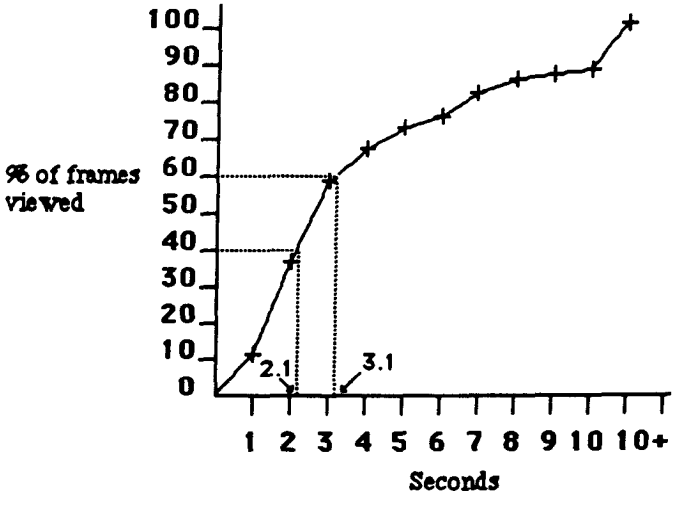



Figure 5.7 Cumulative graph for Lisp AE

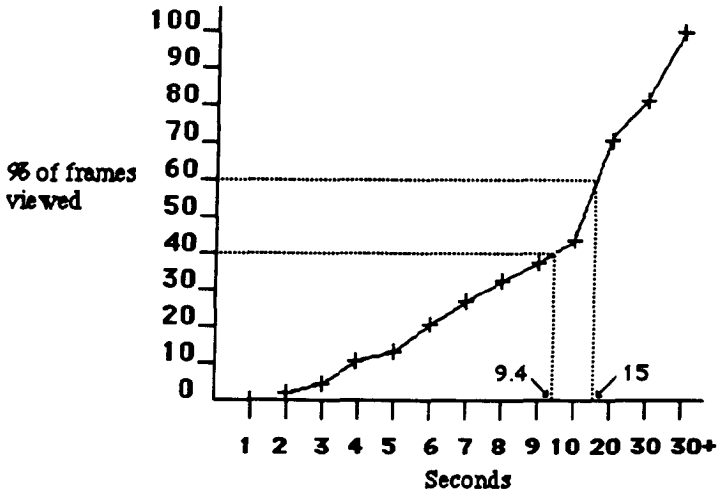



Figure 5.8 Cumulative graph for Lisp IP

Comments from Protocols

**AE:**

On reading the program at the start of the experiment AE understands that the aim of the program is to pass the property of flu. He misses the fact that the program is recursive, and does not fully understand the workings of the program.

*AE: ahm right well it's defining a function called infect which has an argument x er*

*EX: just say generally what you're thinking*

*AE: em.. I presume putprop is going to put something into a database. So the first thing is that when you infect x, x is given the property that x has flu and then err... if er.. if x kisses something then something also gets flu.. em*

*EX: so what's the general idea of the program*

*AE: well you're passing the property of having flu from em x to Mary.*

AE had a problem throughout the stepper display due to having a scant knowledge of Lisp. For example before the experiment the subject did not know how 'putprop' or 'cond' worked, and these had to be explained to him.

AE spotted the recursive nature of the program (when it was being stepped), which he had missed when reading the program at the start of the experiment,

*AE: Ah right so I see, yes em there is a recursion here that I didn't notice, didn't register the first time. Mary's kissed John and it's infect again, we'll see who he infects.*

After the APT-0 display had been seen, AE made the following comments. He thought that the way the display showed that the database did not hold a particular item could have been confusing to someone who had not come across it before, although he understood what was happening himself. When AE had read the program through at the start of the experiment he did not notice that it contained a recursive call. He commented that APT-0 had pointed this out to him when it ran. AE mentioned that although he had problems with the 'cond' clause he thought he would have been able to work out what the 'cond' clause did from the stepper display.

**IP:**

IP appeared to have little understanding of the program before the program was stepped,

*IP: First of all it's defining a function infect and em it's got in it. Well the first one says x has flu and then there is a condition, and I read it as x has flu if and then if the following applies. This one is I'm not sure, the first one says equals is that equals nil.*

*EX: yes*

*IP: get x kisses so that's find out who's been kissing, who's been kissing x and the answer to that is, if the answer to that is nil and then if it isn't true then somebody has been kissing x then you find out em who's been kissing x again. Same question again, and looks like it does the same thing again in fact...*

As in the Prolog version of APT-0 IP immediately recognised the 'edit-time' code in the 'DATABASE' window,

*IP: The database at the top repeats this, so I don't have to look at this any longer.*

When the 'in place subroutine instantiation' occurred IP understood what was happening completely,

*IP: So infect Mary that's calling the function and I have to go forward now so I press f and*

*here it outlines the whole of the function infect x, it says the called Lisp code so it refers back to what's in the database, em so let's see now what's going to happen by going forwards.*

*and because it's called it, it's taken it from the database and put it in the bit I'm working on at the moment. I suppose to the interaction but it hasn't yet done anything with it.*

IP was slightly confused about the presentation of the conditional clause 'cond' - i.e. highlighting the whole 'cond' expression first, then the test, and then the s-expression that would be evaluated first. This is not what he had expected, and confused him as to the order of evaluation in the conditional clause,

*IP: So what's it going to do next? It will try the condition and I should think it will go for the get x kisses as that is the middle most part of that condition.*

*well it does the whole condition first*

*oh Mary has... did I say that.. no I didn't say that so it's gone to equals get x kisses nil rather than get x kisses. Conditional, the test so what does that mean. Finding out who x is from the kisses database.. and see whether it equals nil. I don't know what eq nil would be.*

When new functions were called IP did not know where the brackets came from that get pushed down to the next line (the parentheses from the previous function that have yet to be carried out),

*IP: I don't know, I noticed before the brackets*

*EX: yes those brackets are..*

*IP: are they pointers like arrows*

*EX: they are actually part of the previous procedure, but it would be confusing to put them anywhere else.*

The status line was a great help. It was used a lot when IP was confused about what Lisp was doing, or just needed reassurance about what was happening.

The first comment IP made after the experiment was that the APT-0 display had shown him how recursion works, whereas before he had not understood it. The problem that IP had with the 'cond' clause was, he said, due to misreading the section on 'cond' in a text book. IP thought that the level of detail shown in the APT-0 display was correct, because it showed how you got from one place to another. He mentioned that he found that for each action that he predicted the display showed two or three.

One thing that is very noticeable from IP's protocol is that the first time through the program IP was not too sure what was happening, and went through the display quite slowly. When the program recursed and went through for the second and third times, he was a lot more confident, and made correct predictions all the way through.

Improvements

Both AE and IP viewed category 2 for longer than the marker range. In addition to this, categories 'NO MESSAGE' and 'SIDE-EFFECT' had high viewing times from IP and AE respectively.

Category 'PROCEDURE' deals with procedure calls and is similar in nature to category 'HIGHLIGHT GOAL' in the Prolog version of the stepper. The solution is the same as mentioned previously: a better status line message, which comments explicitly on the actions being taken by APT-0. Category 'NO MESSAGE' represents frames with no status line message. The solution here is to provide a status line message thus providing the subject with information about the state of APT-0.

Category 'SIDE-EFFECT' concerns frames dealing with additions to the database. One point that AE mentioned in his comments is that the way APT-0 showed that the database did not contain the item that was being searched for, could be confusing. The reason for this is that in this case APT-0 highlights those items in the database that are similar to the one being looked for, which is the same as it does

when it finds what it is looking for in the database. The solution to this is not to highlight similar clauses in the database window, and to give a message on the status line that explicitly says that the particular item is not present in the database.

Both subjects spent a long time viewing a few of the frames from categories 'VARIABLE' and 'EVAL S-EXP', as AE did with category 'COND CLAUSE'. Categories 'VARIABLE' and EVAL S-EXP' are those dealing with encountering variables and evaluation of s-expressions, respectively. The process by which variables are selected and bound must be looked at here, as it appears that the status line message is very clear. The problem with the evaluation category is probably due to it being a complex feature of Lisp. I think that the viewing times may be reduced by improving the message on the status line so that users are told specifically what evaluates to what, rather than the generic phrase 's-expression evaluates to ...'. Category 'COND CLAUSE' represents frames showing conditional clauses. I think that the reason frames from this category have caused problems, apart from the fact that the protocols show that both subjects had little understanding of 'cond' before the experiment, is because APT-0 focuses on the whole 'cond' expression and then the test before it evaluates anything. This is not expected by novices and appears to have caused some confusion as to what is going to happen next. There are two solutions to this. The first is not to focus on the whole 'cond' expression and its test, but to move straight to the s-expression that is evaluated first which would be consistent with the way that the rest of the program is evaluated. Secondly the status line message can be made more explicit in informing the user of what is happening.

### *5.2.3 Assembler Prototype*

This section presents the results from the Assembler prototype tracer.

Categories

'NO MESSAGE': Frames that have no message on the status line, apart from denoting which pass the assembler is in. {*Status line*}

'INIT CONSTANTS': Initialisation of constant values, and substitution of mnemonic arguments with these values. The constants are highlighted in the 'edit-time' code, and are inserted into the database. The mnemonic's arguments are highlighted in the 'edit-time' code, and the value is highlighted in the 'trace-time' code. The substitution then takes place. {*Edit = Trace*; *Variable integration*; *Side-effect visibility*}

'NO ACTION': Nothing is done to an instruction on the first pass of the assembler. {*Status line*}

'EVAL CONSTANTS': Evaluation of a mnemonic's argument in the first pass of the assembler. Any constants are substituted for their values (as in 2) and the evaluation of the argument then takes place (i.e. STA PTR1+2). {*Status line*}

'REGISTERS': Removing and adding registers and memory to the database. A message is given on the status line warning of an addition, or removal from the database. Then the instruction is carried out. {*Status line*}

'EXECUTION': Execution of assembly instructions. The instruction is highlighted in the 'edit-time' code, with the corresponding parts of the database being highlighted (i.e. registers, memory). The instruction is then carried out, showing the effect on the flag register. {*Edit = Trace*; *Variable integration*; *Status line*}

'IND ADDRESSING': Indirect addressing of memory using the index registers. This is a complex technique, the exact meaning of which depends on the value held in either the 'X' or 'Y' register. The meaning of this kind of instruction is shown on the status line. {*Status line*}

| **Category** | **Mean time** secs | **Comment** | **Median time** secs | **Comment** |
|---|---|---|---|---|
| NO MESSAGE | 4.7 | OK | 1.1 | OK |
| INIT CONSTANTS | 0.7 | OK | 0.6 | EASY |
| NO ACTION | 1.7 | OK | 0.9 | OK |
| EVAL CONSTANTS | 0.7 | OK | 0.7 | OK |
| REGISTERS | 2.5 | CONFUSED | 1.8 | CONFUSED |
| EXECUTION | 2.3 | OK | 1.0 | OK |
| INDIRECT ADDRESSING | 4.3 | OK | 1.1 | OK |

Figure 5.9 Assembler category data for AE

| Category | Mean time secs | Comment | Median time secs | Comment |
|---|---|---|---|---|
| NO MESSAGE | 4.0 | OK | 1.2 | OK |
| INIT CONSTANTS | 2.1 | OK | 0.8 | OK |
| NO ACTION | 2.5 | OK | 0.8 | OK |
| EVAL CONSTANTS | 0.7 | OK | 0.7 | OK |
| REGISTERS | 6.6 | CONFUSED | 3.0 | CONFUSED |
| EXECUTION | 2.7 | OK | 0.6 | OK |
| IND ADDRESSING | 3.2 | OK | 0.1 | EASY |

Figure 5.10 Assembler category data for MS

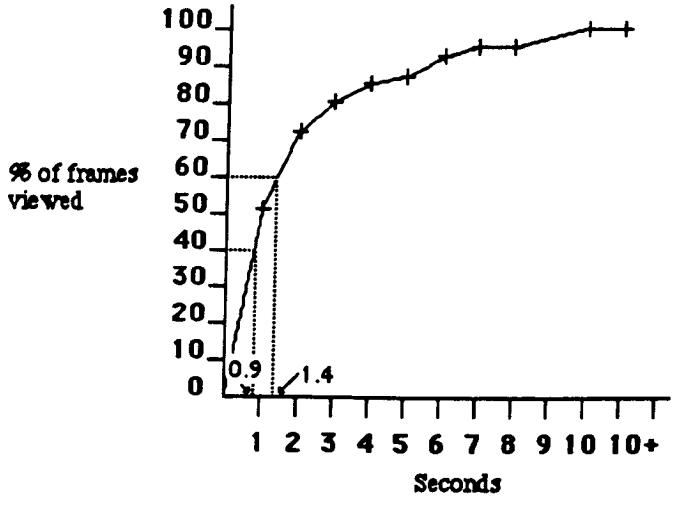



Figure 5.11 Cumulative graph for Assembler AE

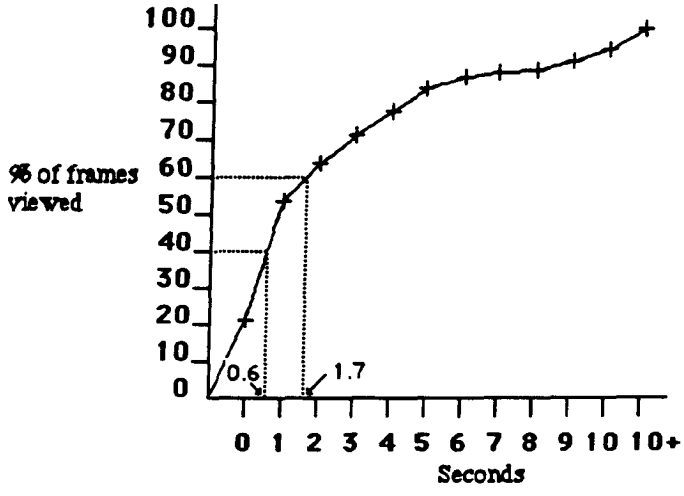



Figure 5.12 Cumulative graph for Assembler MS

## Comments from Protocols

**AE:**

AE had few problems with the display and predicted correctly what was happening all the way through, apart from the action of indirect addressing,

*AE: Oh I see so it displays what the equivalent instruction on the bottom line here of what LOAD A 10,Y evaluates to now that Y actually has a value in it.*

*but it doesn't show on there, fair enough*

*so it's 12 so it looks in address 12*

*and it's got a value in there*

*eh, does an add again expanded at the bottom, and it's adding the contents of an address to the contents of Y*

*EX: What that is actually doing is adding the contents of address 50 plus Y*

*AE: Let me go back*

*EX: so it's the contents of, the address is 50 + Y*

*AE: Oh, sorry yes. and it adds that to the accumulator. I made an assumption that, I was probably confused by other assemblers.*

There was some confusion about the meaning of the word mnemonic, in the first banner page. He thought that the word referred to the assembler mnemonic rather

than the name of constants. This meant that AE thought that the assembler mnemonic was going to be replaced with machine address values, which he would not have wanted to see,

*AE: em I wasn't sure what you meant in the instructions when you said mnemonic I assumed you meant the...*

*I wasn't to clear what you were going to do, I thought surely he's not going to put in the machine code values.*

The only comment AE made after the experiment was that after seeing the program stepped by APT-0 he had a clearer understanding of what the program did.

**MS:**

MS made correct predictions throughout the APT-0 display, and made extensive use of the status line message.

MS did not agree that that the second part of the stepper should show the execution of the code. He felt that another stage should have occurred before this,

*MS: second part execution of the code. No I wouldn't agree that that comes next, I would have thought there is assigning.*

*removing assignment mnemonics, I wonder what that means, right*

*that's obvious what that means. Right so we are moving into execution now. I don't think thats right because, I think another part of the assembly process should have been to replace the labels blockadd and next, well should have assigned addresses to all the code, and replace the labels with those addresses, that's the parallel process to replacing the symbolic constants that I've just seen anyway.*

MS thought that the address spaces should hold a random value on power up rather than nothing as this is more realistic,

*MS: Actually I seem to remember Tim making this point, thats not very clever because it implies that memory can have no contents, whereas in fact it can't. There is bound to be something in there, random rubbish or something.*

He only noticed the status line message about a third of the way through the stepped program,

*MS: and ah for some mysterious reason on the bottom line, which it hasn't shown me previously before, it's given me a hint of what the known value of Y which I can see up there.*

MS did not like the scrolling of the memory boxes to display the relevant parts of memory (which happened due to the lack of screenspace),

*MS: I found that suprising, cause on seeing the grey band move down, I thought that it was the notion of the band moving was to draw my attention to something different, and I found it confusing there to see the band moving, and it's really the scenery that's going past.*

The comments that MS made following the experiment where all about the story that the APT-0 stepper had told. MS on the whole did not like the APT-0 representation of the way 6502 Assembly language worked.

He felt that it did not tell the truth of what was happening. I have already talked about the 'truth' the model of a language portrays, and I think that part of the problem is that MS was not a novice but a language/systems implementor. This meant that the model of the language he wanted to see was at a lower level than the stepper display was presenting. MS thought the presented model of the Assembler should have been a more correct representation of the action of an Assembler, with the mnemonics and labels being replaced with memory addresses, and two passes of the Assembler before execution of the Assembly code. MS also brought up the notion of showing addresses and mnemonics together for greater clarity, this is the way that Assembly programs are usually printed when listed on paper.

Improvements

Both subjects viewed frames from category 'REGISTERS' for a longer period than the marker range. This group of frames concerns removing and adding registers/memory to the database. This is an unusual process and there is a lot of information to take in at this part of the display. A high viewing time here then is not totally unexpected, and may well not be a problem after the first time the stepper is used. Again a more informative status line message may improve the situation.

Both subjects also spent a long time viewing a few of the frames from categories 'NO MESSAGE', 'NO ACTION', 'EXECUTION' and 'IND ADDRESSING', while MS alone spent a long time viewing category 'INIT CONSTANTS'.

Category 'NO MESSAGE' represents frames that have no status line message. The solution here as with Prolog and Lisp, is to provide a message commenting on the action of APT-0 during these frames.

Category 'NO ACTION' deals with frames that show that nothing happens when particular instructions are executed in the first part of APT-0. The answer to this problem is to modify the status line message which at present says 'part1: - no change'. This tells the user that no change occurs, but not why it is that nothing happens. A more informative message is needed, for example 'part1: - there are no constants to substitute in this instruction'.

The 'EXECUTION' category concerns the execution of Assembly instructions. Long viewing times might be expected in this category, because when an instruction is executed a lot of side effects happen to the database. The flag register is changed, and the index register and/or memory is involved. In comparison with other categories there is a lot of information for the user to read and assimilate, which will produce a higher viewing time.

Category 'INDIRECT ADDRESSING' deals with indirect addressing using the index registers. This is again quite a complex process, and one might expect some long viewing times, while subjects take in what APT-0 is showing them. I think that little improvement can be made here.

Category 'EVAL CONSTANTS' represents the evaluation of an assembler mnemonics argument. An improved status line message detailing what is being evaluated and the result of the evaluation should make these frames easier to understand.

In his comments at the end of the experiment AE pointed out that the use of the word 'mnemonic' in part 1 is confusing, as it refers to the constants used in the program, whereas in Assembly programming the normal meaning of 'mnemonic' is the assembly instruction, i.e. 'STA', 'LDA'. This problem can be solved by replacing the word 'mnemonic' with the word 'constant'. MS did not like the way APT-0 represented the action of the Assembler program. He suggested either replacing the mnemonics with machine code addresses, or displaying both the mnemonic and the address side by side as in Assembler listings. If the mnemonics were replaced with an address the resulting display would loose a lot of meaning for novice programmers. The latter suggestion, displaying both mnemonic and address, would be more appropriate allowing the novice to see a meaningful and truthful picture of program execution. I think the most important point here is not necessarily telling the correct story of what happens when an assembler program runs, but to tell a story that is understood by novices, and works in terms of the language they are used to programming in.

## 5.3 Discussion

It must be noted that all the users understood perfectly at the end of the experiment the action of the programs which the stepper displayed to them. Only MS on the 6502 display had a full understanding of the action of the program before it was stepped. The other subjects had some understanding of the program, but the details were not understood. Features like recursion were at first missed by AE and IP in both the Prolog and Lisp programs, and the order of people that got the flu was not known.

The display of recursion did not give anyone any problems. On the contrary it seemed to clarify the concept for IP who did not understand it, and pointed out the recursion to AE who did not spot it in the program. When the APT-0 display was viewed AE spotted that recursion was happening. More interestingly IP, who had a

poor knowledge of recursion (he did not really know the difference between iteration and recursion), spotted that there was something strange happening in both the Lisp and Prolog programs, but did not know it was recursion until he was told.

Looking at the comments more specifically, it appears that the status line was found to be very useful, especially for IP and AE, when the subjects were confused or had lost track of what was happening. The verbal protocols show that the status line was referred to continually by both AE and IP for information or confirmation of what the display was showing. The ability to move forwards and backwards in the display was found useful by all the subjects, who used it occasionally from time to time to clarify a point that was confusing.

The level of detail shown in the stepper appeared to be about right for the subjects IP or AE. However it would be nice to make this feature flexible for the more adept user.

There were several apparent problems with the stepper displays. The content of the status line was sometimes confusing. The messages used in the APT-0 displays were very abrupt, sometimes using programming terms which novices might not understand. The message must be more explicit, and refer to the contents of the frame specifically. In other words instead of a general message like 'evaluation', the message should inform the novice about what it is that is being evaluated, and what it evaluates to. This approach should provide the novice with enough information to understand the display, but if not it should tell the user enough to point him/her in the correct direction to find out.

The first part of the Assembler version of APT-0 caused some confusion due to the usage of the term 'mnemonic', which the display was using to mean constant. The real meaning of 'mnemonic' in Assembler refers to the instructions, i.e. 'STA', 'LDA'. This was easy to put right by a simple substitution of the word 'constant' for the word 'mnemonic'.

The display of those instructions in a program which are still waiting to be carried out could be the cause of some confusion. This was exemplified by IP not knowing what was the meaning of the brackets that were left over from the previous procedure (Lisp) during recursion. Showing that items are not present in the database has produced some difficulty and has been presented differently.

More general changes that have been made to improve the presentation of information in the APT-0 display have to do with the window titles and the status line messages. The title 'DATABASE' is perfect for the Prolog display as the database is what the top window shows. It is however a little strange for the Lisp and Assembler displays, and has been changed accordingly. The Assembler top window is called 'MEMORY', and the Lisp top window 'LOADED FUNCTIONS/S-EXPRESSIONS'. These titles communicate the content of the windows more clearly to the novice than the previous titles. The status line used in APT-0 give rather abrupt and general messages to the user. This has been changed so that the messages are more specific to the frame being shown, and are less abrupt, thus making it easier for the user to understand the more complex parts of the stepped program.

In summary the following five lessons have been learned from the experiments presented in this chapter.

1) All stepper frames must have a status line message commenting on the display.

2) The message contained in the status line must be explicit, and refer specifically to the information contained by the frame. A general message is not sufficient for novice programmers.

3) Care must be taken to ensure that terms used in the status line message have one meaning only. If they have more than one connotation then they become confusing, and loose their usefulness as a method of communication.

4) Do not impose a structure on the presentation of information to the user. For example in the Lisp prototype the structure of the 'COND' function was shown before it was evaluated. This can destroy the consistency of the display, and result in confusion.

5) The method used for showing the absence of information in the display, should not consist of highlighting the information that is present. This is not consistent with the use of highlighting. The status line should inform the user that a particular piece of information is not present.

The above five lessons together with the design principles, discussed in chapter 3, and the Prolog prototype have been used as the basis for the design and implementation of a real animated Prolog tracer. This tracer, which is presented in the next chapter, allows the design principles to be used to show the execution of all the features in the Prolog language, rather than the few displayed in the prototype.

# CHAPTER 6

# APT

## 6.1 Why Prolog

Although the prototypes of APT were developed for three different languages, and the design principles upon which the prototypes were built are meant as general principles for any programming language, it was necessary to choose one language for which a real animated stepper could be built and tested. Prolog was chosen for several reasons which I will elucidate below.

Firstly Prolog contains some very complex concepts which are difficult for novices to learn as they are not intuitive. The reason for this is that the concepts involved such as backtracking, unification, the cut and side effecting are processes which have no common equivalent in everday life, whereas the equivalent of procedure calling such as in Lisp is a more common occurrence.

Secondly, Prolog also has a lack of tools available to the user to enable him/her to understand the process by which the Prolog interpreter works, and so has a great need for the type of animated tool that APT provides.

## 6.2 The Implementation

The APT system is written entirely in Lisp, including the Prolog interpreter, and runs on the Symbolics 3600™ family of machines under release 6 of the system.

The implementation is based on the design principles presented in chapter 3 and follows them rigorously. Two of the design princples have been excluded from the design of the system due to lack of time, but they could be added to the system without affecting the design. The two principles are firstly the ability to move forwards and backwards in the animated display, which requires a history of the display to be recorded; and secondly the integration of the interpreter's error messages into the trace-time code. The interpreter which was specifically built for APT (described below) contains no error system , and so error messages cannot be integrated into the trace-time code.

The sections below describe how each of the components of APT (see figure 6.1) is implemented. The description of each component is at a high level and is not concerned with language and machine specific implementation details unless necessary.

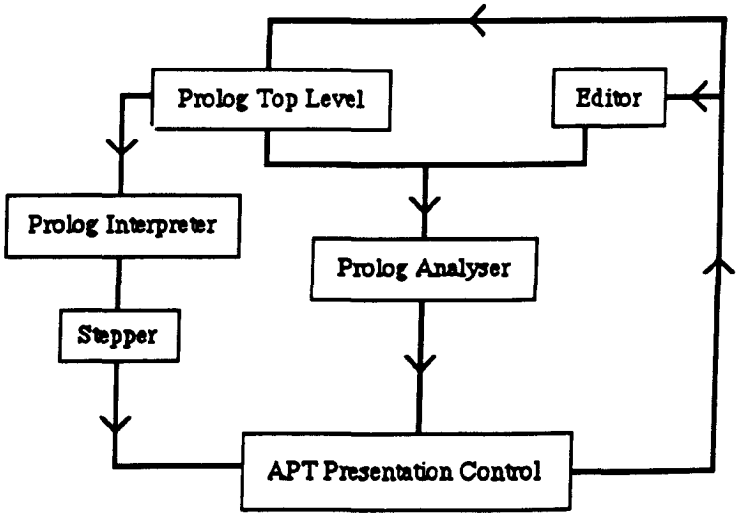



Figure 6.1 The Architecture of APT

### *6.2.1 The Prolog Interpreter*

The Prolog interpreter is written in Common Lisp, and comprises the majority of the recognised Prolog features, including the cut and list manipulation. The syntax differs slightly from the standard Edinburgh syntax, and more closely resembles that of MacProlog™, although it is possible to alter the syntax under user control. The main differences between Edinburgh syntax and APT syntax are as follows: ':-' is written as 'if' and the ',' used for conjunctions of goals is written as 'and', the commas between arguments are optional. The last difference is that variables are preceeded by an underscore '_' rather than by a capital letter as in Edinburgh syntax.

A purpose-built interpreter for APT was required for several reasons. The most important of these was because none of the commercially available Prolog systems allow the kind of access to the detail of program execution necessary to build APT. Secondly few machines that run a commercially available Prolog have the flexibility, and environmental support necessary to build an animated stepping system of the complexity of APT.

### *6.2.2 The Stepper*

The stepper is embedded within the interpreter, and produces stepped information both for APT and for the more traditional teletype display. The stepper is invoked by the command 'step' which remains in force only for the next Prolog query. So, if the user wishes to step a query, the command 'step' must be given prior to the query each time.

The stepper is split into four sections within the interpreter, each section providing information pertaining to a salient feature of the Prolog interpreter. These four sections are called step-act; step-nil; step-next and step-cut. Respectively they deal with:

a) attempting to unify the current goal to a database entry

b) the failure of the above attempted unification

c) backtracking and the attempt to unify the current goal to the next database entry

d) hitting the cut on backtracking

**Step-act** provides information concerning the current goal, the candidate chosen from the database which the interpreter is attempting to unify with the goal, the current environment which contains the variable bindings and the level of the goal. The query posed to Prolog is the top-level goal and is always level zero. Each new subgoal that Prolog attempts to solve increments the level by one. The level of the goal being solved is necessary information because the variable bindings held in the environment are indexed using this level. So in order to determine the instantiated versions of the goal and candidate both the environment and the level are needed. In simple terms step-act tells us what the current goal is, the database entry it is being unified with, and the value of any variables in the goal and candidate.

**Step-nil** is concerned with when all the possible candidates in the database have been exhausted by the resolution process, or in other words when the current goal has failed. The information provided is the failed goal and the associated variable bindings.

**Step-next** deals with backtracking. When backtracking occurs and alternative solutions are required for the goal backtracked to, step-next provides the following information: the new goal that has become the current goal, all the possible candidates that it might unify with, and the associated variable bindings at that time.

Lastly, **step-cut** deals with the problem of hitting the cut on backtracking. The only information provided here is the candidate from the database that has failed due to the cut.

As a supplement to the information presented by the above processes a record of each unified goal and associated details is kept in the form of a Lisp object (for more

information on Flavors and objects see Weinreb and Moon 1981). This object contains the following information; the name of the goal (bound and unbound), its subgoals (surface and internal representation), the variables bound at that level, any shared variables, the indentation level needed for the display, and the call level of the interpreter. The name of the object is the level of the goal that is unified. Figure 6.2 shows a typical object of this type and the contents of its slots.

| | |
|---|---|
| NAME: | "kisses(_X,fred)" |
| UNBOUND-NAME: | "kisses(_X,_Y)" |
| SUBGOALS: | NIL |
| INTERNAL-SUBGOALS | NIL |
| VARIABLES-BOUND-AT-THIS-LEVEL: | ((_X JANE)) |
| BINDINGS: | ((_Y FRED) (_X JANE)) |
| LEVEL: | 3 |
| INDENT: | 5 |

Figure 6.2 A typical object and its contents

These objects contain the name of the goal, as a string, that has been unified in both its instantiated form (NAME) and uninstantiated form (UNBOUND-NAME). If this goal is a Prolog rule the object will contain these subgoals in the form of a string (SUBGOALS), and in the internal format (INTERNAL-SUBGOALS). This particular object is a fact and so has no subgoals. Other information in the object are the variables that have been instantiated using this rule/fact (VARIABLES-BOUND-AT-THIS-LEVEL), so that these variables can be uninstantiated on backtracking; the variables that have been instantiated in the program so far; the level of the invocation of the interpreter (LEVEL), and the level of indentation of the rule/fact in the trace-time code (INDENT).

### *6.2.3 The Prolog Top Level*

The top-level is a psuedo top-level in the sense that it resides in an ordinary editor window that may be scrolled and edited in the normal way. The top-level of the Prolog tracer can be seen in the screen snapshots presented in the scenario later. It is labelled 'Prolog Window' and resides in the lower half of the screen.

Queries are entered into the top-level by means of a 'do-it' key, in this case the 'END' key. When the 'do-it' key is pressed the Prolog query is read and passed on to the interpreter to be executed. Any output that is generated by this query is then printed out to the window just as if it were a top-level window.

Not only is this pseudo top-level a useful feature because it allows the user editing and recording facilities, but it is essential for the purposes of APT. As will be described later in this section, for APT to work it is necessary that the presentation control knows where Prolog output and input is. In other words the position of Prolog queries and output must be known in order to manipulate the text under program control. It would not be possible to record the position of the Prolog text if it did not reside in an editor buffer.

### *6.2.4 The Editor*

The editor in which Prolog rules and facts may be witten, is an ordinary editor buffer allowing normal editing facilities such as scrolling, deleting, inserting and marking regions for manipulation. The editor of the Prolog tracer can be seen in the screen snapshots presented in the scenario later. It is labelled 'Editor Window' and resides in the top half of the screen.

The editor is where as in most programming environments the user writes programs before executing them. As in the Pseudo top-level besides the facilities provided by the normal editing commands an important feature is the access an editor

buffer provides to the positions of Prolog text within the buffer. This allows the analysis of the user's program described in the next section, and the highlighting of the edit-time code with inverse video.

### *6.2.5 The Prolog Analyser*

The aim of the analyser is to pinpoint the positions of the sub-components of Prolog clauses in the editor buffers of the Editor and Prolog windows. The components of a Prolog clause are the variables, constants and arguments contained within the clause, this may include list structures, plus the higher level structures such as the predicate and subgoals of the clause, and lastly the whole clause itself. It is necessary for APT to know the positions of these components so that when the animation is in progress particular items such as variables can be picked out of the buffer and highlighted using inverse video.

The analyser is used in three places during the animation of a program. The first is to determine the position of the user's edit-time code contained in the editor window. This is carried out each time the 'step' command is given to ensure that the information generated by the analyser is correct at the time of animation. As the data contained in the database and the editor window are always the same there are no inconsistencies between the program held in the database and the program held in the Editor window.

The second place that the analyser is used is in processing the query that has been entered in the Prolog window. This is needed so that the variables and constants given in the query can be referenced when the first goal is matched. The analyser is only called once at the time the query is asked.

Lastly, the analyser is used to keep track of the components of the growing goal tree generated in the Prolog window. Every time an alteration is made to the animated display, whether it be instantiating variables, uninstantiating variables, insertion of

subgoals or removal of subgoals due to backtracking or the cut, the analyser is called to update the information held on the positions of the sub-components of the growing Prolog clause.

The process by which the analyser parses Prolog clauses is the same for all three cases. Each clause is parsed using the Prolog syntax to delimit each component of the clause. For example each clause is delimited by the full stop at the end, the start is the first piece of text after a full stop. The predicate is all the text that preceeds the 'if' delimiter. Subgoals follow the 'if' and are split by the key word 'and', apart from the last subgoal which ends with a full stop. Lists are parsed using square brackets to denote the start and end, embedded lists are parsed in the same way. Arguments are seen as being anything following an opening parenthesis until a comma or closing parenthesis is found. Complex clause structures are parsed in the same way using the above methods over and over as embedded structures are processed.

APT stores all the information generated by the analyser as objects. The components of each Prolog clause being stored as a separate object which has the name of the clause itself for access (figure 6.3). So for example the clause

```
like(_X,_Y) if
  has(_Y,money) and
  kisses(_X,_Y).
```

would be parsed into the object below and have the name

|like(_X,_Y) if has(_Y,money) and kisses(_X,_Y).|

The slots of the object, seen in capital letters, are the labels by which the information held in each category is accessed.

NAME: "like(_X,_Y) if has(_Y,money) and kisses(_X,_Y)."

START-BP: ("like(_X,_Y) if" 0)

END-BP: (" kisses(_X,_Y)." 15)

HEAD: ("like(_X,_Y)" ("like(_X,_Y) if" 0) ("like(_X,_Y) if" 11))

SUBGOALS: (("has(_Y,money)" (" has(_Y,money) and" 2) (" has(_Y,money) and" 15)) ("kisses(_X,_Y)" (" kisses(_X,_Y)." 2) (" kisses(_X,_Y)." 15)))

ARGS: (("_Y" (" kisses(_X,_Y)." 12) (" kisses(_X,_Y)." 14)) ("_X" (" kisses(_X,_Y)." 9) (" kisses(_X,_Y)." 11)) ("money" (" has(_Y,money) and" 9) (" has(_Y,money) and" 14) ("_Y" (" has(_Y,money) and" 6) (" has(_Y,money) and" 8)) ("_Y" ("like(_X,_Y) if" 8) ("like(_X,_Y) if" 10)) ("_X" ("like(_X,_Y) if" 5) ("like(_X,_Y) if" 7)))

CONSTANTS: (("money" (" has(_Y,money) and" 9) (" has(_Y,money) and" 14))

VARIABLES: (("_X" (" kisses(_X,_Y)." 9) (" kisses(_X,_Y)." 11)) ("_Y" (" kisses(_X,_Y)." 12) (" kisses(_X,_Y)." 14)) ("_Y" (" has(_Y,money) and" 6) (" has(_Y,money) and" 8)) ("_X" ("like(_X,_Y) if" 5) ("like(_X,_Y) if" 7)) ("_Y" ("like(_X,_Y) if" 8) ("like(_X,_Y) if" 10)))

EDITOR-WINDOW: #<window 7776765>

Figure 6.3 A typical object containing clause information

The meanings of most of the slot names are obvious due to their names, but to prevent any confusion they stand for the following: NAME is the name of the clause that this information pertains to; START-BP and END-BP are the buffer pointers to the start and end of the clause; HEAD consists of the predicate and its buffer pointers; SUBGOALS contains each subgoal of the clause and their buffer pointers; ARGS is all the possible arguments contained in the clause and their buffer pointers (some of the arguments will be the same as the elements of the variables and constants slots because of ambiguity); CONSTANTS holds each of the constants found in the clause together with their start and end buffer pointers; VARIABLES is the same as constants but for the clauses variables; and lastly the EDITOR-WINDOW holds the value of the window that the clause resides in (this is necessary for the highlighting process).

The information held in each slot of the object may look rather daunting but will become clear with the explanation that follows. Apart from the information held in the editor-window slot, which holds the internal name of the window that the clause is found in, there are only two types of structure. Firstly there is the string, which is anything bounded by double quotes i.e. "string". Secondly there is the buffer pointer. A buffer pointer is one of the structures used in the Zmacs editor to keep track of the text contained within a buffer. All a buffer pointer consists of is a string containing some text, and a number which is an index of the string denoting where in the string the buffer pointer points to. The string and number are themselves contained within a list or parentheses i.e. ("like(_X,_Y) if" 0) is a buffer pointer pointing to the start of the string at the beginning of the list.

The object holds both buffer pointers to the start and end of each component of the clause. With this detailed positional information it is possible to pick out any component for highlighting.

### *6.2.5 APT Presentation Control*

The 'presentation control' of APT consists of three parts; the messages that are shown on the STATUS LINE; the insertion and deletion of text in the trace-time code; and the highlighting of the relevant pieces of text in both the edit-time and trace-time code. The cumulative effect of the different parts of the 'presentation control' results in the animated view of program execution that the user sees.

It should be pointed out that in this version of APT variables are not renamed during the display of the execution of programs. This will cause problems at certain points in the execution of programs ie recursive calls. However the design of APT does not prevent this feature being added at a later date.

Figure 6.1 shows that the 'presentation control' receives information from two sources. The 'stepper' provides information concerning the state of program

execution, i.e. which goal is being unified against which clause, the value of any variables present. The 'analyser' on the other hand provides information concerning the state of the display of program execution, i.e. what text is displayed, and the exact position of that text on the screen. This information allows the presentation control to know which piece of code has to be manipulated at each step of program execution, and where that code is on the screen.

The rest of this section describes what happens on the screen for each type of event that occurs in the execution of Prolog programs. In the description below goals, variables and values they are printed in <angle brackets>, rather than by specific names, and 'message' refers to the 'staus line' message. Examples of many of these descriptions can be seen either in the scenario below or in Appendix G, H or I. Where this happens a comment points out where the example can be found.

*Query/goal unification:*

If there is a possible match for the goal in the database (a clause with the same predicate) both the goal in the trace-time code and the clause in the edit-time code is highlighted. If the clause is a rule only the head is highlighted. See Fig. 6.5.

message: Trying to match <goal> against <database entry>

If unification is not possible because the database contains no clauses of the same predicate the following status line message is given:

message: No more definitions of the predicate <name> in the database

*Unification succeeds:*

If the goal unifies against the head of a rule the whole rule is highlighted in the edit-time code. A copy of the rule is taken from the edit-time code and the body of the rule is inserted in the trace-time code directly below the goal, which corresponds to the head of the rule. If the goal is the query then both the head and body of the rule is

inserted below the query (see Fig. 6.6). This allows the user to be able see the query in its original state, with any variables remaining unbound.

message: Match succeeded - trying rule

If the goal unifies with a fact and the goal contains no variables then the display moves on to the next goal, because the unification is trivial.

message: Match succeeded

*Variable instantiation:*

When a goal is unified, and it contains variables the following three steps happen:

1) Both the goal and the clause it has unified against remain highlighted. A message warns the user that variable instantiation is about to occur (see Fig. 6.9).

message: About to instantiate variables in the subgoal

2) The highlighting of the goal, and clause is removed. The instances of the variable to be instantiated in the trace-time code are highlighted, together with the value in the edit-time code which the variable is about to be instantiated to (see Fig. 6.10).

message: The variable <name> is matched against <value>

3) The highlighting of the variable in the trace-time code is removed, the variable deleted and replaced with the value it has become instantiated to. The value is then highlighted (see Fig. 6.11).

message: The variable <name> is instantiated to <value>

*Backtracking:*

The following actions occur when backtracking is initiated (see Appendix H).

1) When a goal has been matched against one or more clauses but fails to unify, the clause that was last attempted remains highlighted (edit-time code) along with the goal (trace-time code).

message: Backtracking:- no more definitions of the predicate <name> in the database

2) The highlighting of the previous goal, and its attempted match is removed. The goal being backtracked to is highlighted in the trace-time code.

message: Backtracking to <goal>

If the goal backtracked to is a Prolog primitive which is not retried on backtracking the following status line message is given in place of the one above:

message: Backtracking to <primitive> which is not retried

*Uninstantiation of variables on backtracking:*

When backtracking happens, as described above, it is sometimes necessary to uninstantiate the variables contained in the goal that is backtracked over. This happens when variables have been instantiated in the goal when that goal was unified last. The following two stages describe what happens (see Appendix H):

1) The goal in the trace-time code remains highlighted, and the status line warns the user of the impending action.

message: Backtracking:- about to uninstantiate variables which were bound here last time round

2) The highlighting is removed from the goal, and all the relevant instances of the value which is bound to the variable to be uninstantiated are highlighted in the trace-time code.

message: Backtracking:- about to uninstantiate <variable> which is instantiated to <value>

3) The highlighting of the relevant values is removed. The values are deleted and replaced with the variable to which it was instantiated. The instances of the variable are then highlighted.

message: Backtracking:- uninstantiating <value> back to <variable>

This is repeated until all the necessary variables have been uninstantiated.

*Failed rules in backtracking:*

When backtracking happens and rules fail, other rules with the same predicate name are tried. This means that the body of the old rule must be removed from the display to allow room for the body of the new rule to be displayed and executed. The following stages describe this (see Appendix H).

1) When a goal fails backtracking is initiated (see section 2 of backtracking above). If the goal that is backtracked to is the head of a rule then the whole rule fails. The goal being backtracked to is highlighted in the trace-time code.

message: Backtracking to goal

2) If any variables needed to be uninstantiated it would happen here. See section 'Uninstantiation of variables on backtracing' above.

3) The head and body of the failed rule are highlighted in the trace-time code.

message: Backtracking:- about to remove the subgoal/s of <goal> which have failed

4) The highlighting of the body of the rule is removed, and it is deleted from the trace-time code. The head of the rule remains highlighted in the trace-time code, and an alternative match is sought.

message: Trying to match the goal <name> against <clause>

*The cut:*

The following stages describe what happens when during the execution of a program the 'cut' is encountered on backtracking (see Appendix I). The first stage shows the initial unification of the cut.

1) The 'cut' trivially matches.

message: Trying to match the goal ! against ! which is a primitive

2) The head and body of the rule in which the cut resides is highlighted in the trace-time code

message: CUT encountered: parent goal <name> and all its subgoals fail

3) In the same way that variables must be uninstantiated on backtracking as mentioned above, variables must be uninstantiated here if they were instantiated when the head of the rule (or subgoals of the rule preceding the cut) was unified. The highlighting of the rule in the trace-time code is removed. The instances of the value which are instantiated to the relevant variable are highlighted.

message: CUT encountered: first must uninstantiate <variable> which is instantiated to <value>

4) The highlighting of the value/s is removed. The value/s is deleted from the trace-time code and replaced with the variable it was instantiated to. The instances of the variable are highlighted in the trace-time code.

message: CUT encountered: uninstantiating <value> back to <variable>

5) The highlighting of the variable is removed. The rule containing the 'cut' is highlighted in the trace-time code.

message: CUT encountered: about to remove the parent goal and subgoal

6) The highlighting of the body of the rule is removed. The body of the rule is then deleted from the trace-time code. The head of the rule remains highlighted.

message: Backtracking to <goal>

7) If the head of the rule that contained the 'cut' is part of another rule then by default that rule also fails. However an alternative match for the head of this higher level rule may be tried. The head of the rule that contained the 'cut' remains highlighted in the trace-time code. The removal of the failed goals happens in the same way as described in the above section 'Failed goals on backtracking'.

message: Backtracking:- about to remove the subgoal/s of <goal> which have failed

## 6.3 A Scenario

This section will discuss the outward appearance of APT when the system steps through a program, rather than the internal mechanisms that have been described above. What you are about to see are a series of snapshots of APT in action. This is not the best way to present this system. The whole point of such a system is that because dynamic features are hard to understand in a static representation they should be shown dynamically i.e. as an animation. So trying to present an animation system statically is going to cause problems. It will lose a lot of its power of communication, but hopefully it will convey a flavour of the approach. The following scenario consists of the first ten steps in the execution of the program. A complete listing of snapshots showing the complete execution of the program can be seen in Appendix G. No comments are provided to describe what is happening at each step in the programs execution because the message on the status line, along with the associated inverse video highlighting adequately explains what the snapshot is displaying.

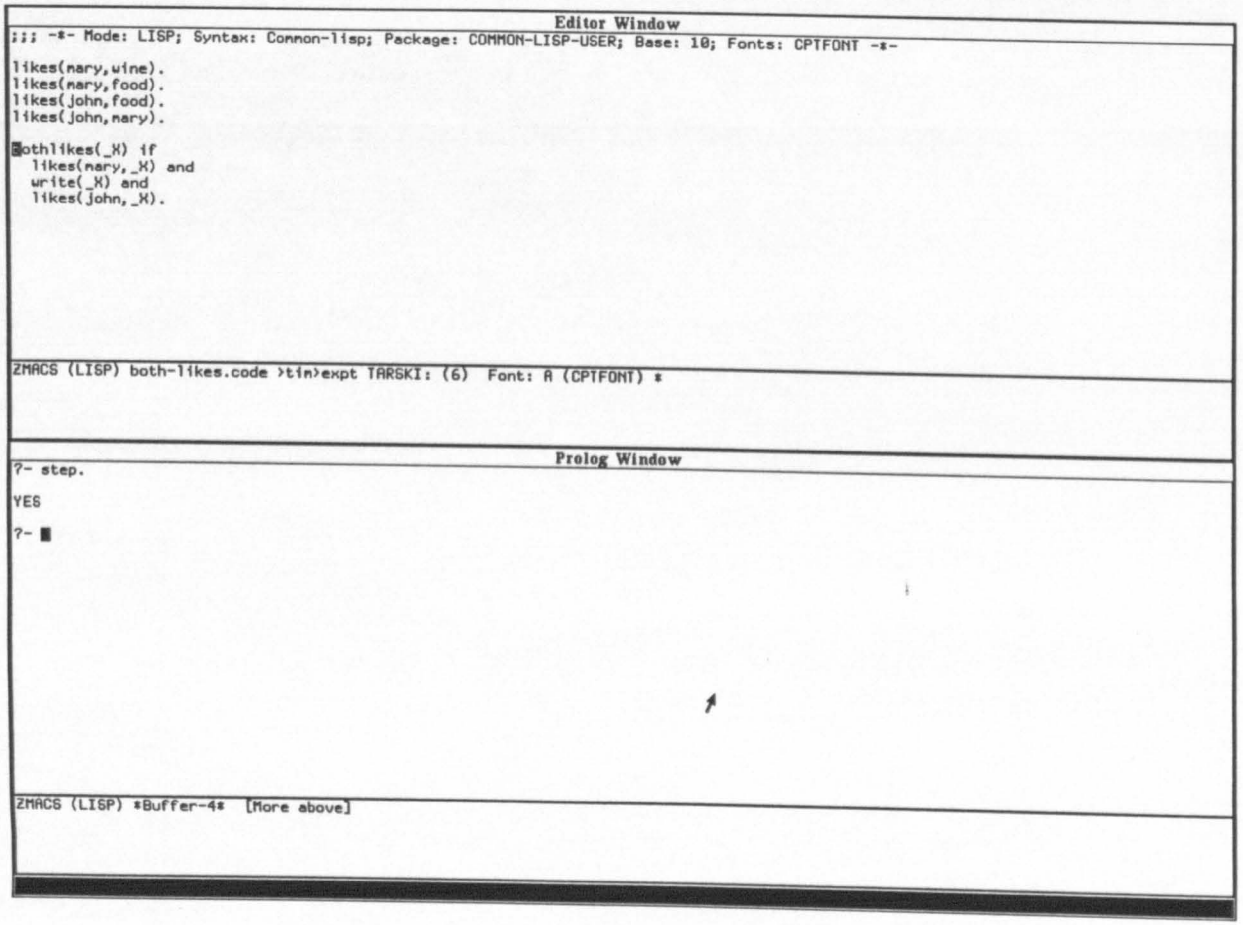



Figure 6.4

A series of screen snapshots from APT

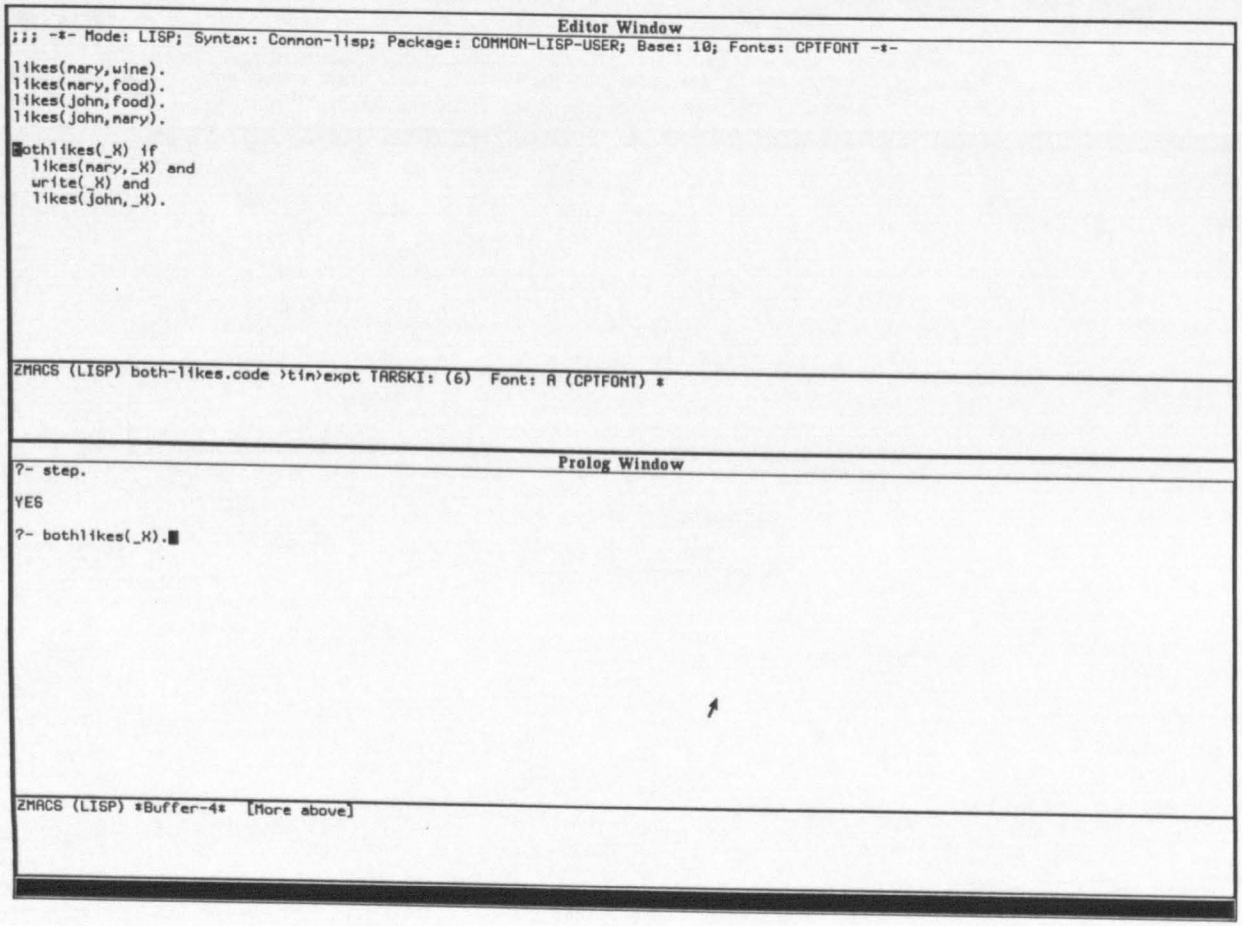

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tin>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
ZMACS (LISP) *Buffer-4* [More above]

Figure 6.5

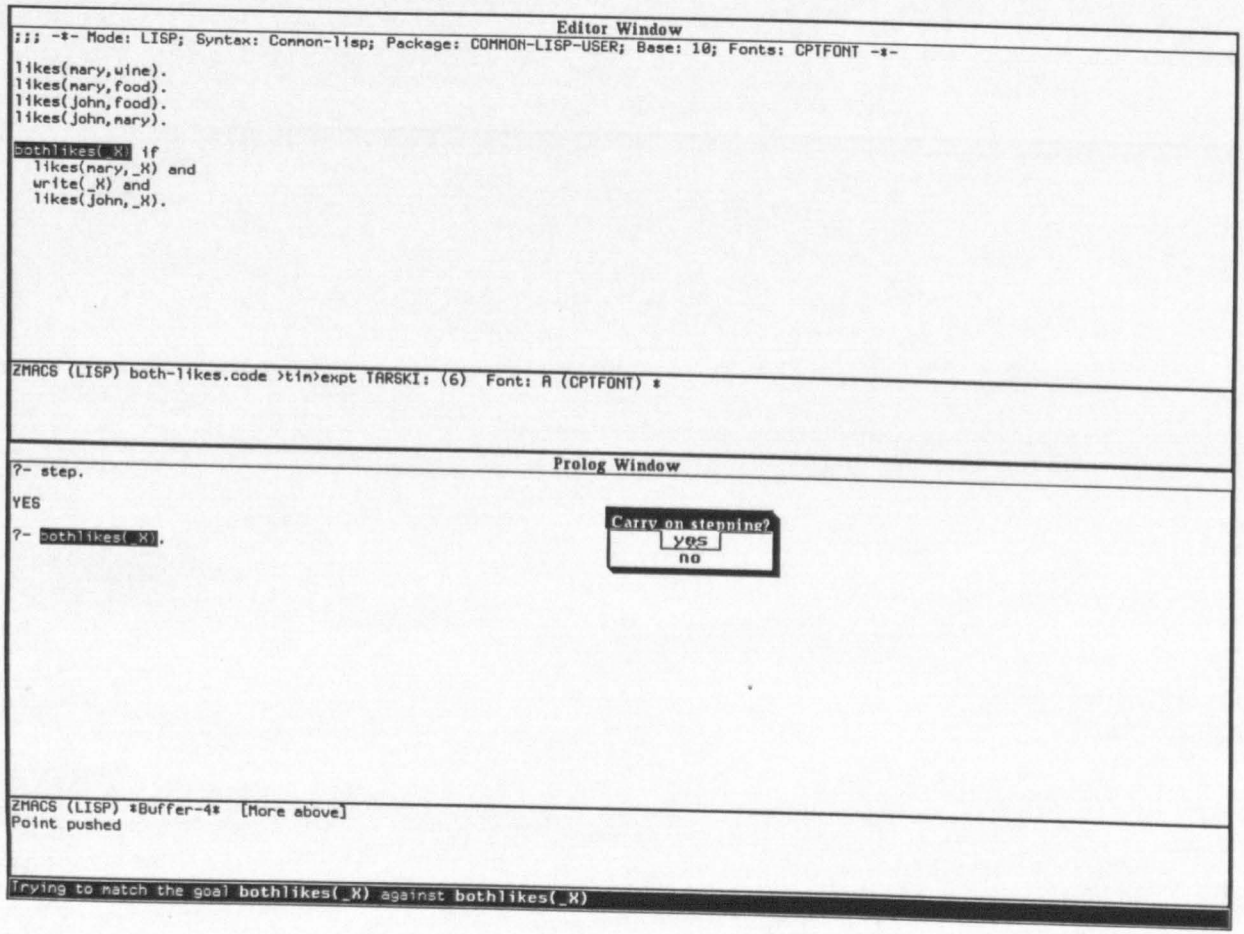

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tin>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
Trying to match the goal bothlikes(_X) against bothlikes(_X)

Figure 6.6

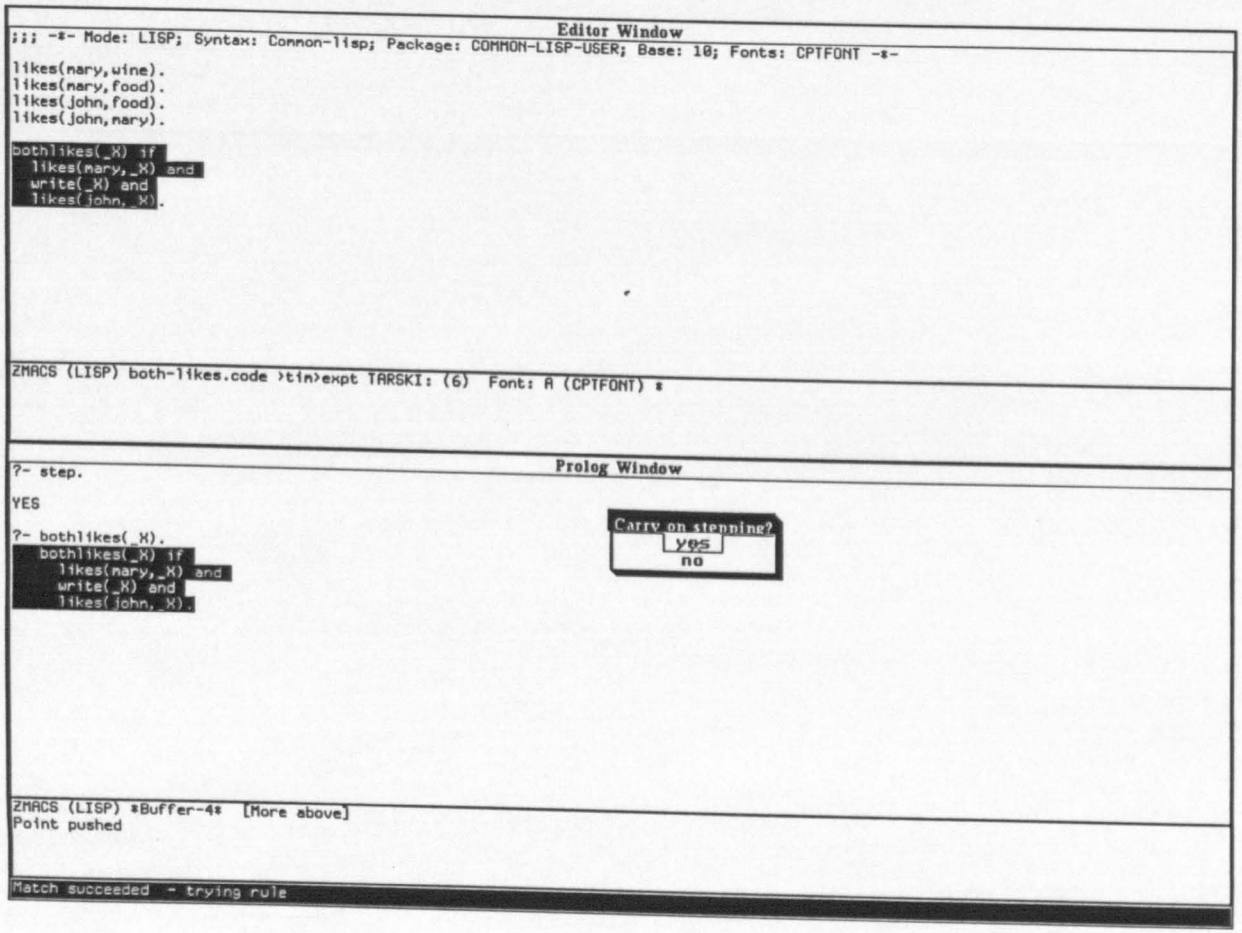

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tim>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
Match succeeded - trying rule

Figure 6.7

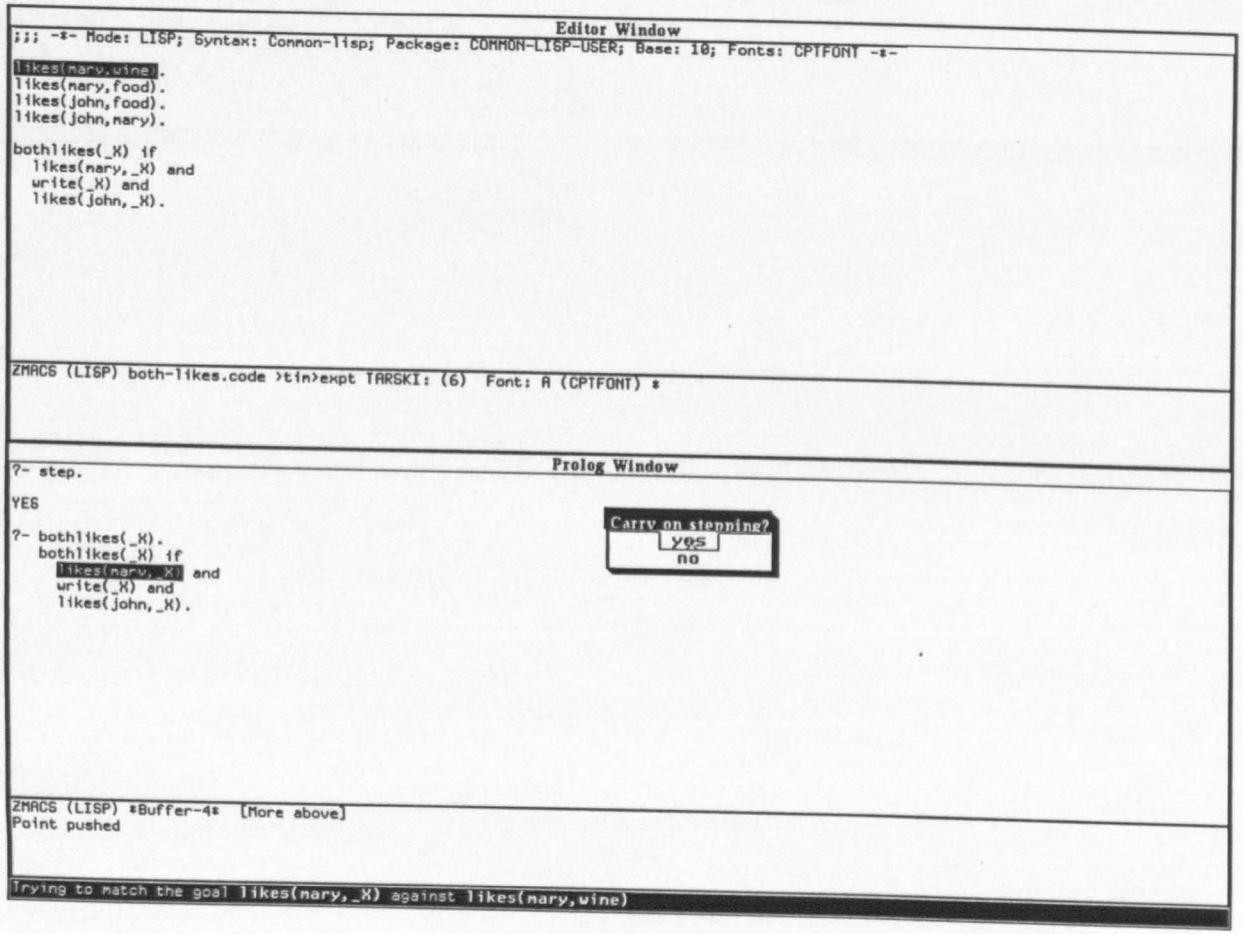

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tin>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
Trying to match the goal likes(mary,_X) against likes(mary,wine)

Figure 6.8

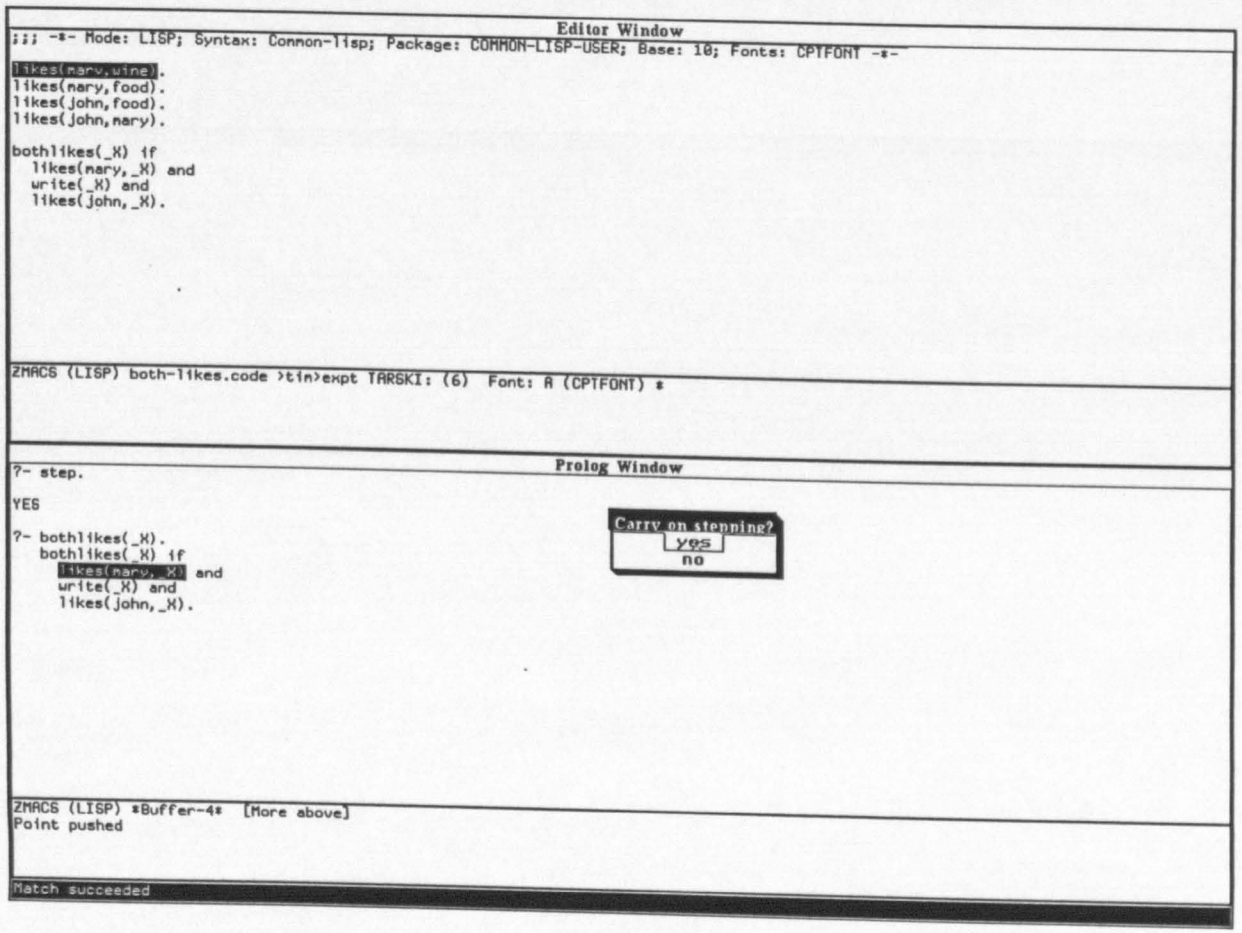

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tin>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
Match succeeded

Figure 6.9

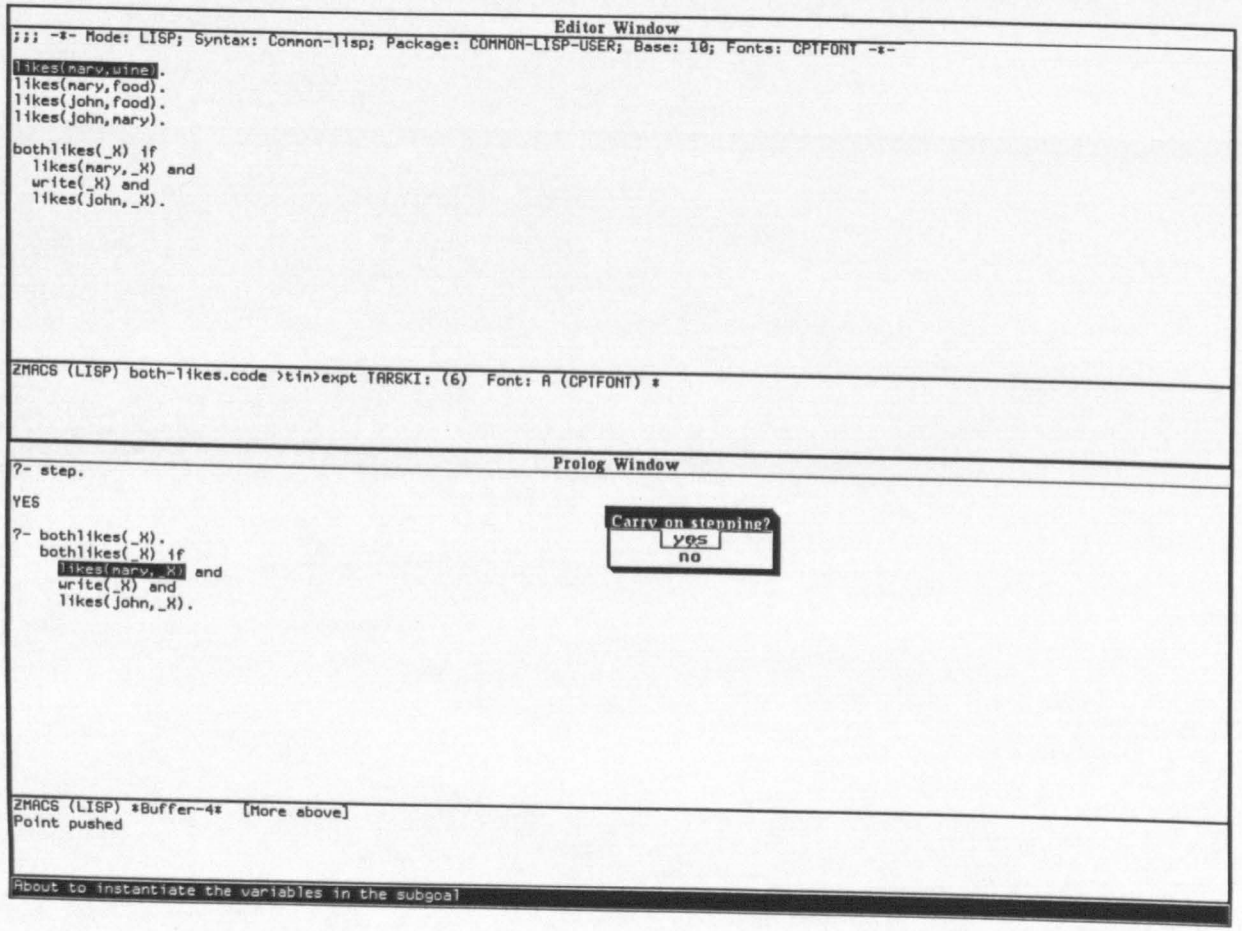

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tin>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
About to instantiate the variables in the subgoal

Figure 6.10

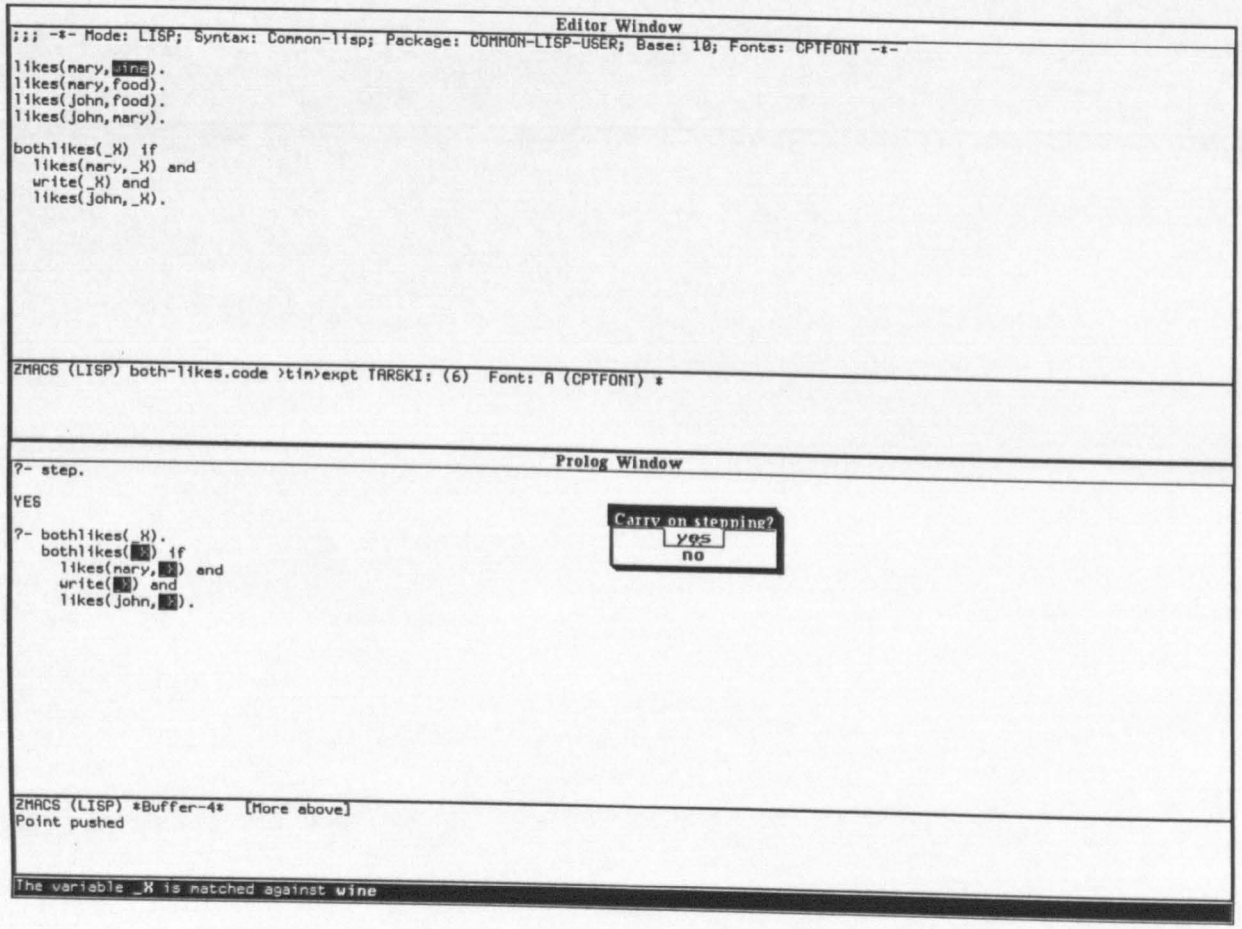

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tim>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _X is matched against wine

Figure 6.11

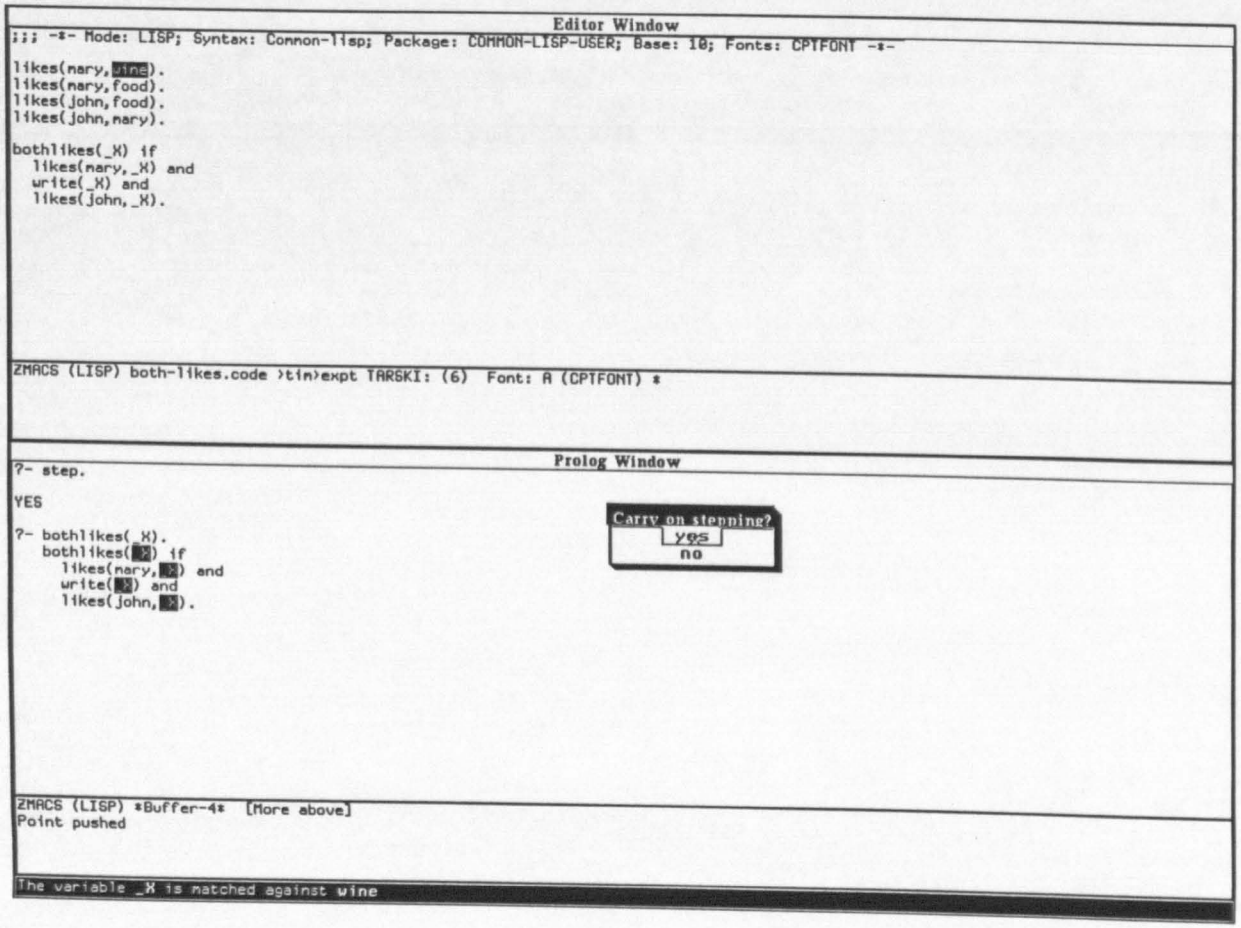

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tin>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _X is matched against wine

Figure 6.12

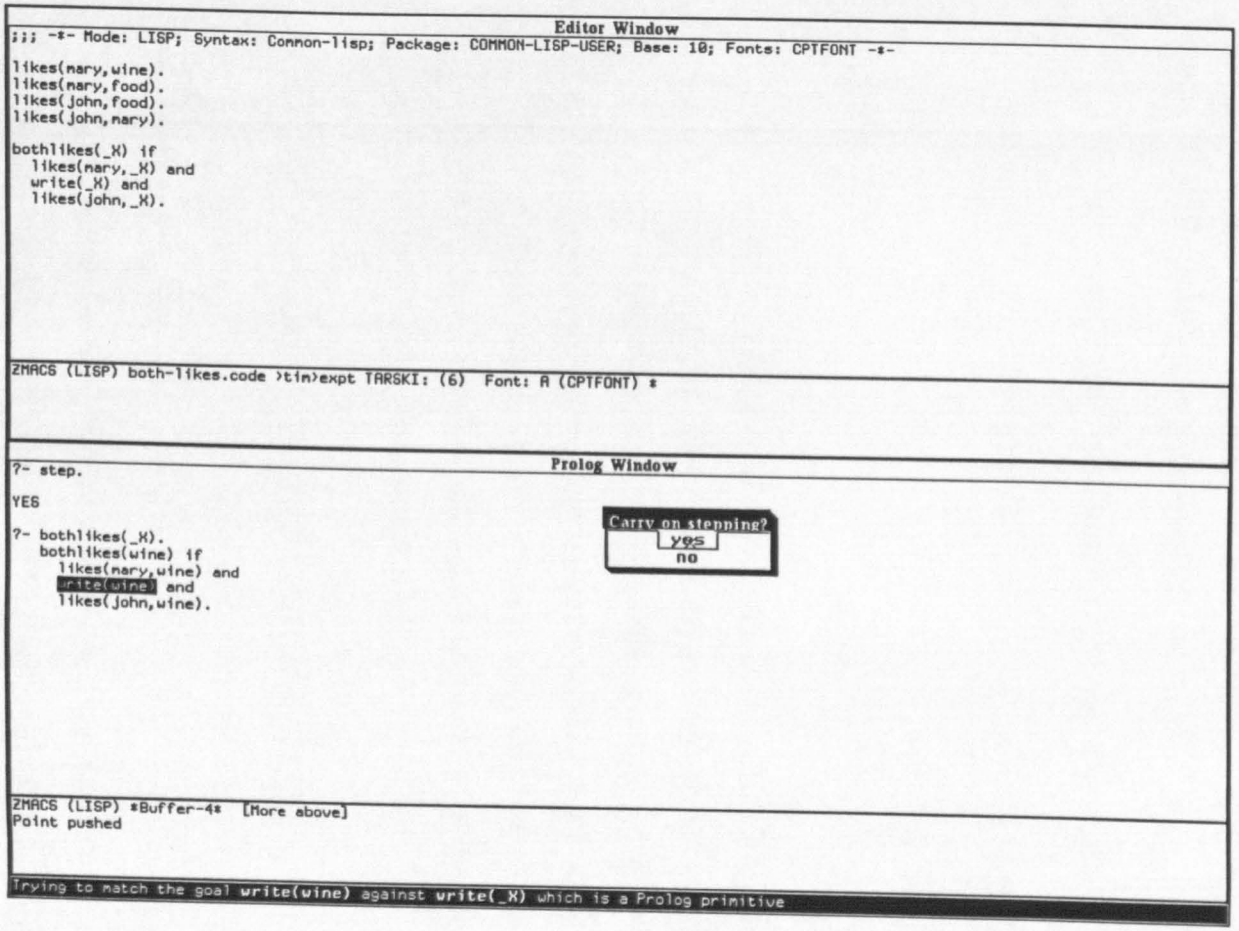

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >tin>expt TARSKI: (6) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
Trying to match the goal write(wine) against write(_X) which is a Prolog primitive

Figure 6.13

# PAGE

# NUMBERING

# AS ORIGINAL

*List Processing:*

The following stages describe what happens when during the execution of a program which includes list manipulation (see Appendix K). In nearly all cases unification of lists work in the same way as any other Prolog atom.

For example these Prolog structures unify to give the following instantiations:

| Structure 1 | Structure2 | Instantiations |
|---|---|---|
| [b] | _X | _X = [b] |
| [a,b] | [_X,_Y] | _X = a _Y = [b] |

However lists can be represented in two different ways, (i) as above, ie [a,b,c] or (ii) using bar notation, ie [_X|_Y]. Bar notation gives a different meaning to the contents of lists. In essence it splits the list into two parts. The part in front of the bar is the head of the list, and the part following the bar which is the tail of the list. So [a,b,c] is a list containg the items 'a' 'b' and 'c', while [a | b,c] is a list whose head is 'a' and whose tail is the list [b,c]. A list therefore can be represented in many different ways. The following lists are equivalent:

[a,b,c] = [a | [b,c]] = [a | [b | [c]]] = [a | [b | [c | []]]]

[a] = [a | []]

Normally the bar notation is implicit in the list and it remains unseen. The list structures above show the representation of lists when the bar notation is used explicitly. Note the use of the empty list '[]' in the last example.

Bar notation is a useful technique when writing programs that manipulate list structures. Take for example the program 'append' which takes two lists and combines them to form another list.

```
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
```

Thus, the query 'append([a],[b],_What)' gives _What = [a,b].

The two different list representations cause a problem when APT traces programs which have both ordinary lists and lists containing bar notation. It means that lists have to be translated from one representation to the other when variables are instantiated with values in the trace time code. This is illustrated in appendix K which shows a full listing of APT stepping through a call to 'append'. Extracts from this listing are shown below which present aspects of list unification/instantiation in APT.

Figs 6.14 - 6.17 show the two lists '[_X|_L1]' and '[a,b]' being unified and the resulting instantiations of variables '_X' and '_L1' in the trace time code. This produces the list '[a | [b]]' which is equivalent to '[a,b]'.

Figs 6.18 and 6.19 show the unification/instantiation of the lists '[_X|_L1]' and '[b]'. Because the list '[b]' can be represented as '[b | []]' it unifies against '[_X|_L1]' giving the instantiating '_L1' to the empty list '[]'.

Figs 6.20 and 6.21 show how items are combined to build up new lists. The variable '_L3' in the structure '[a | _L3] gets instantiated to '[_X | _L3]' which produces the list '[a | [_X | _L3]].

Figs 6.22 and 6.23 again show new lists being built up. Here the variable '_L' in the structure '[a | [b | _L]] becomes instantiated to the list '[c]' to give the new list '[a | [b | [c]]]. This is equivalent to the list '[a,b,c]'.

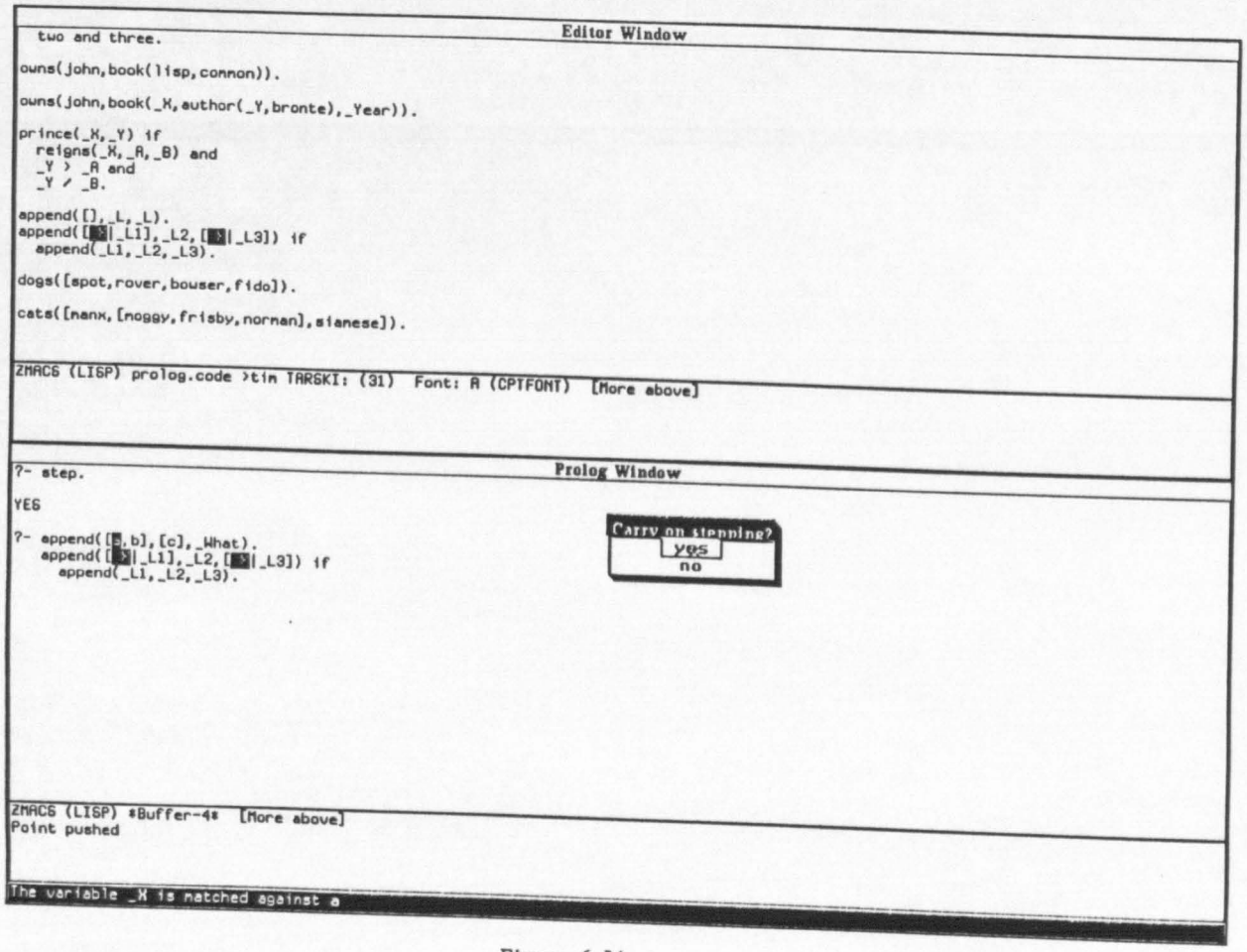



Figure 6.14

A series of screen snapshots from APT - a scenario of list manipulation

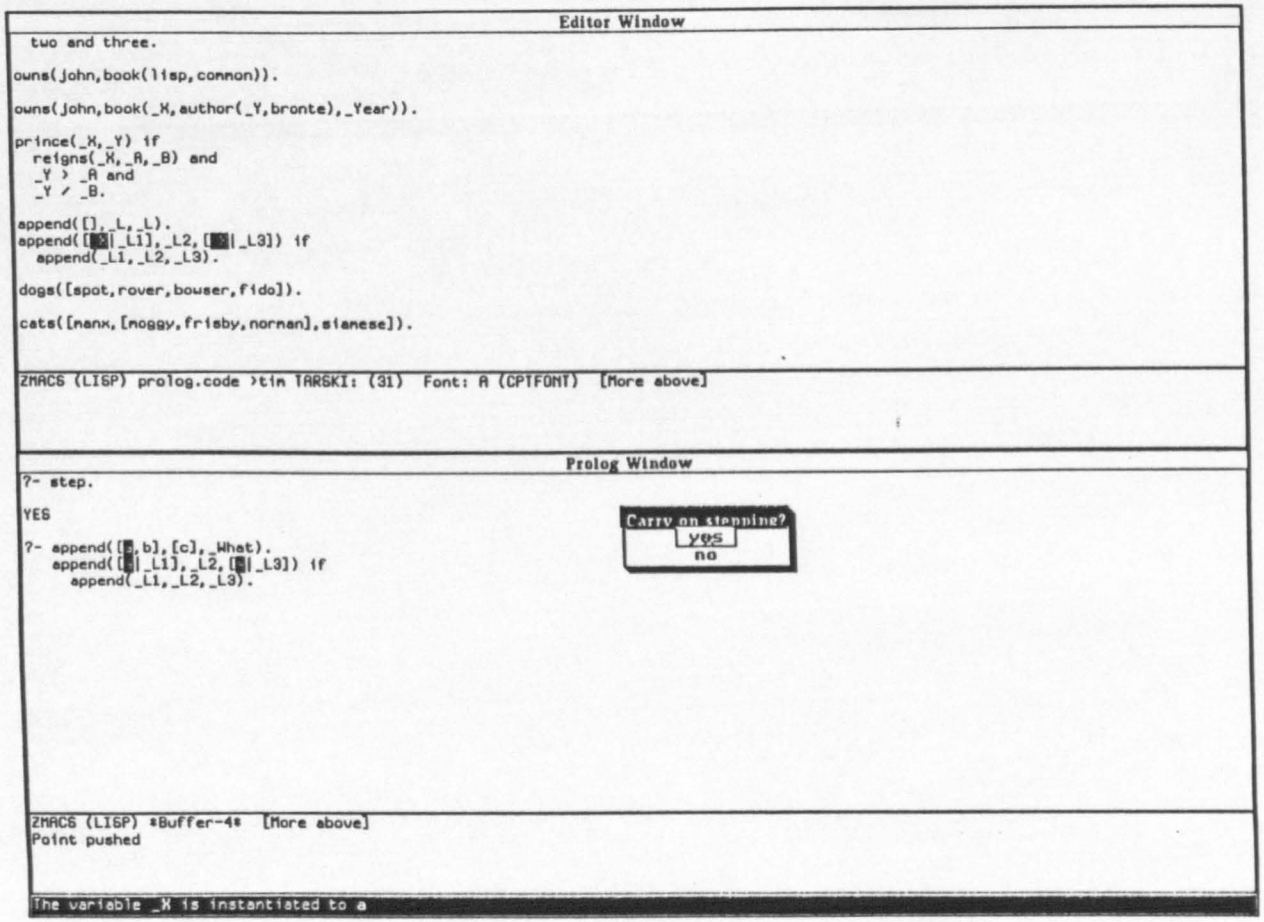

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _X is instantiated to a

Figure 6.15

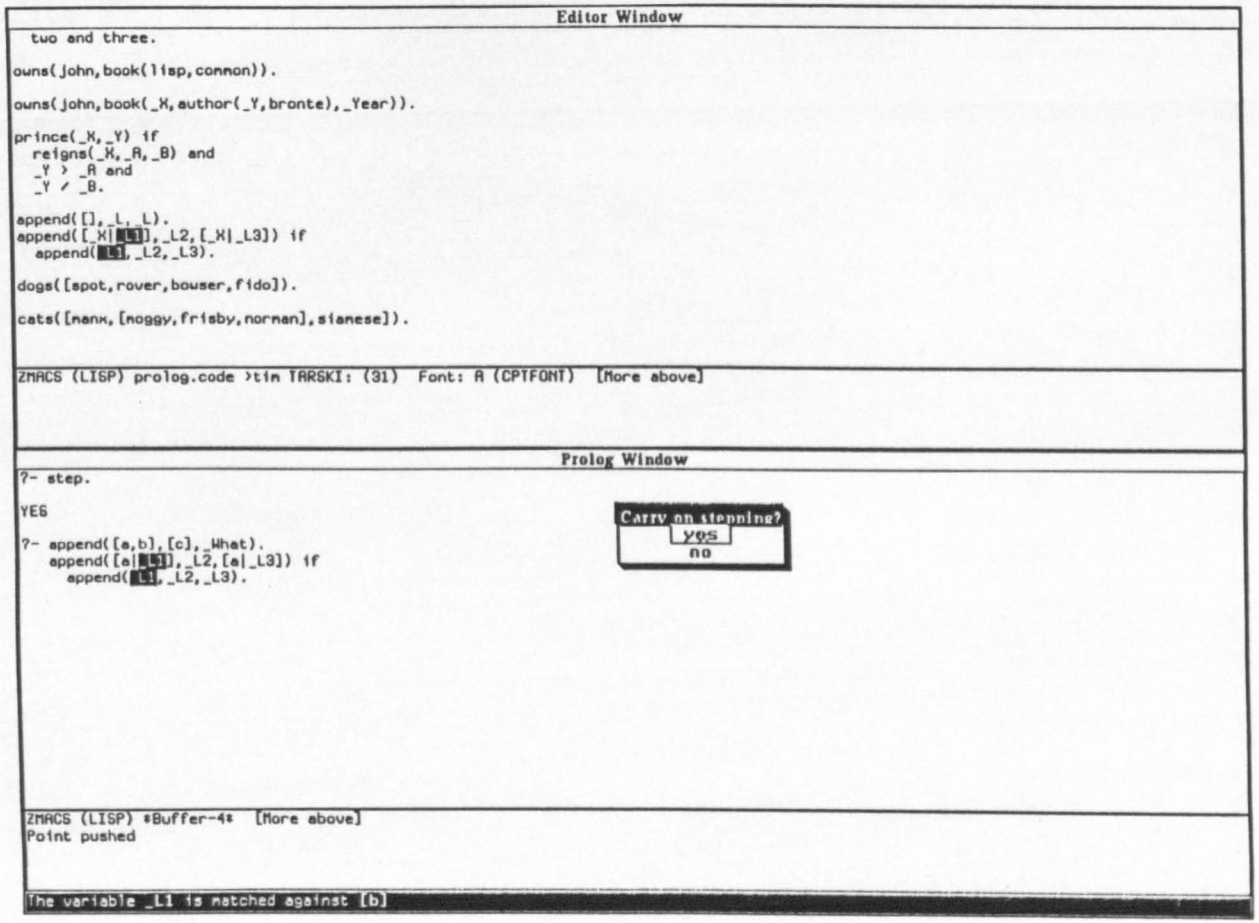

Editor Window
two and three.
owns(john,book(lisp,conmon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|_L1],_L2,[a|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _L1 is matched against [b]

Figure 6.16

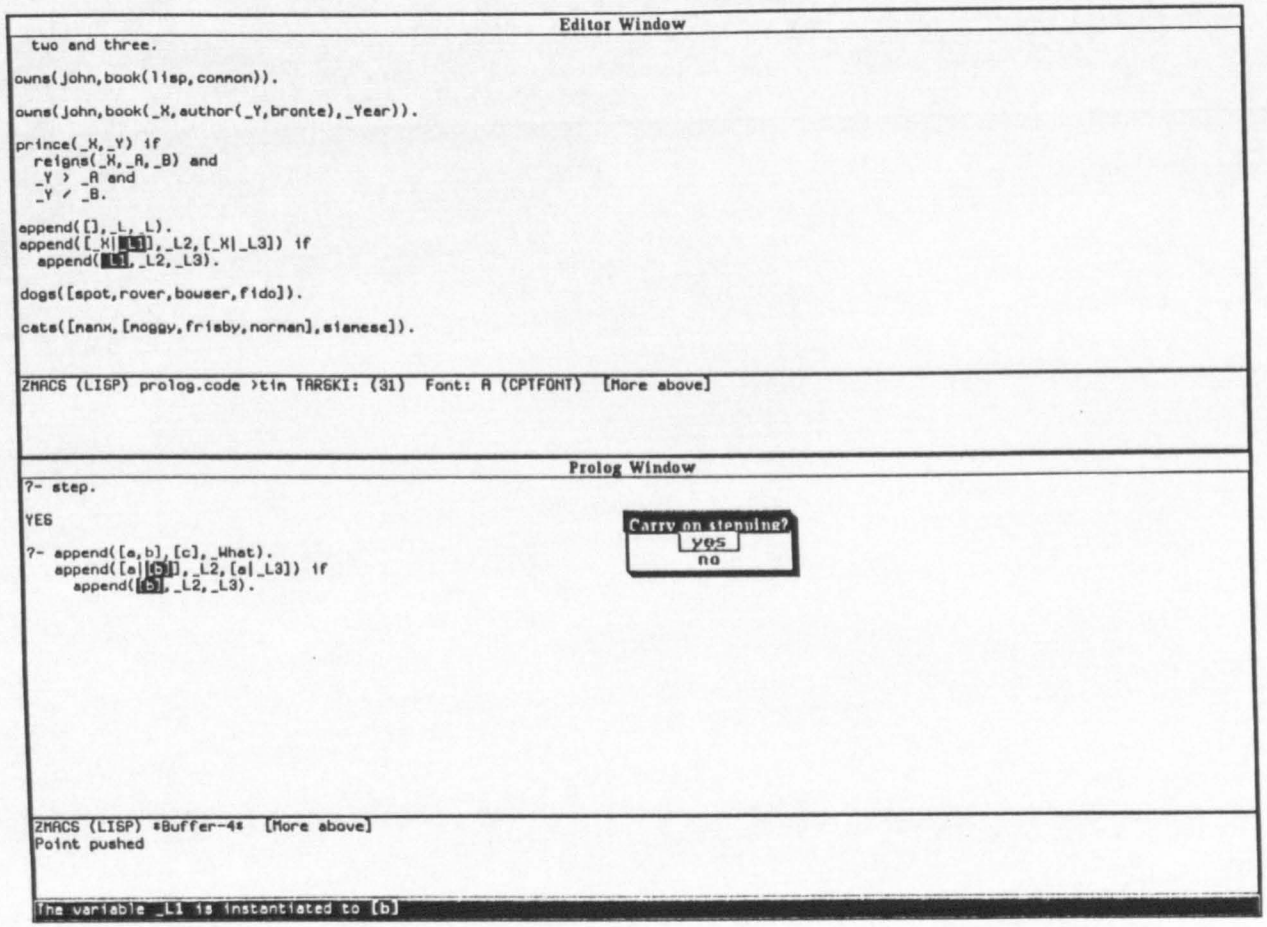

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],_L2,[a|_L3]) if
append([b],_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _L1 is instantiated to [b]

Figure 6.17

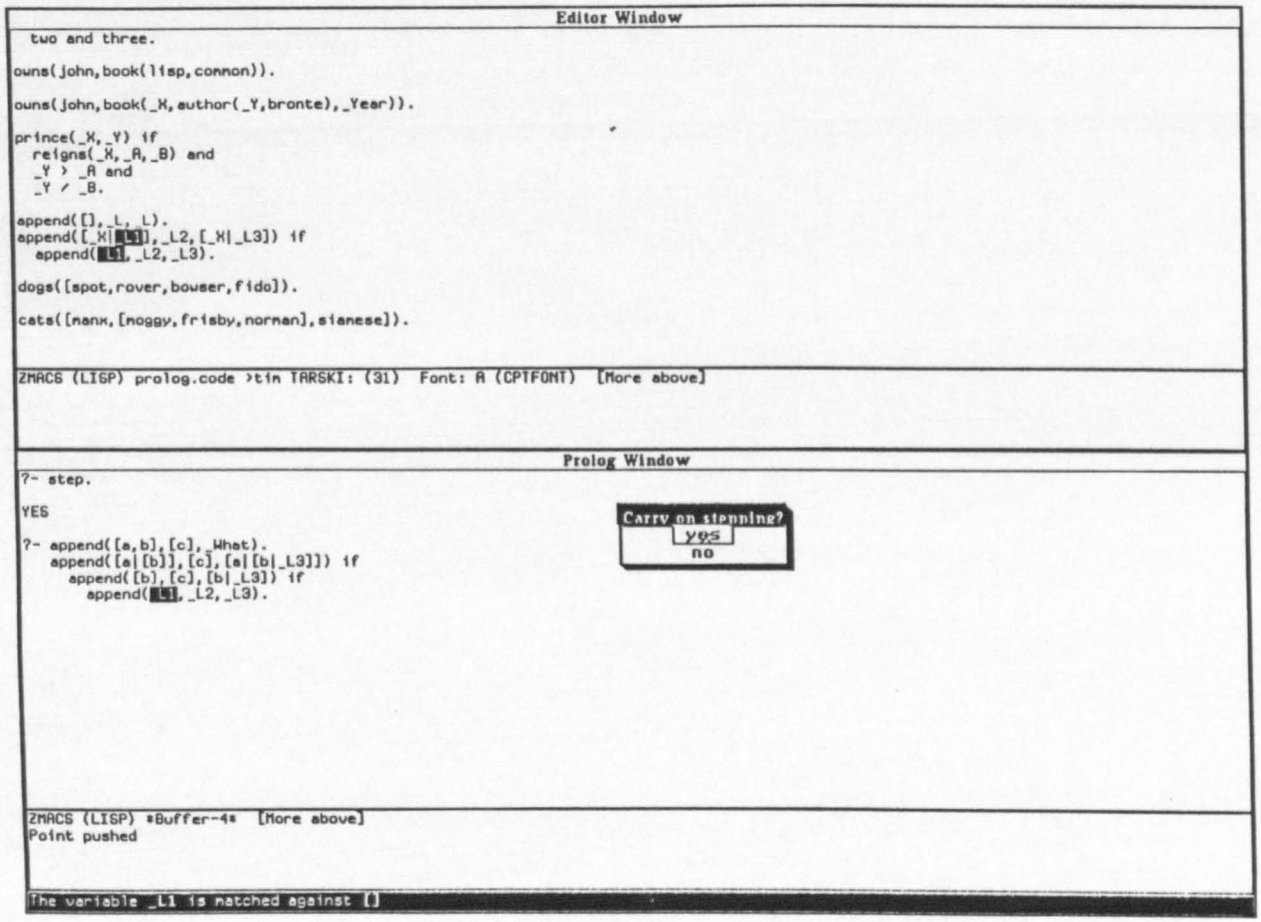

Editor Window
two and three.
ouns(john,book(lisp,common)).
ouns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _L1 is matched against []

Figure 6.18

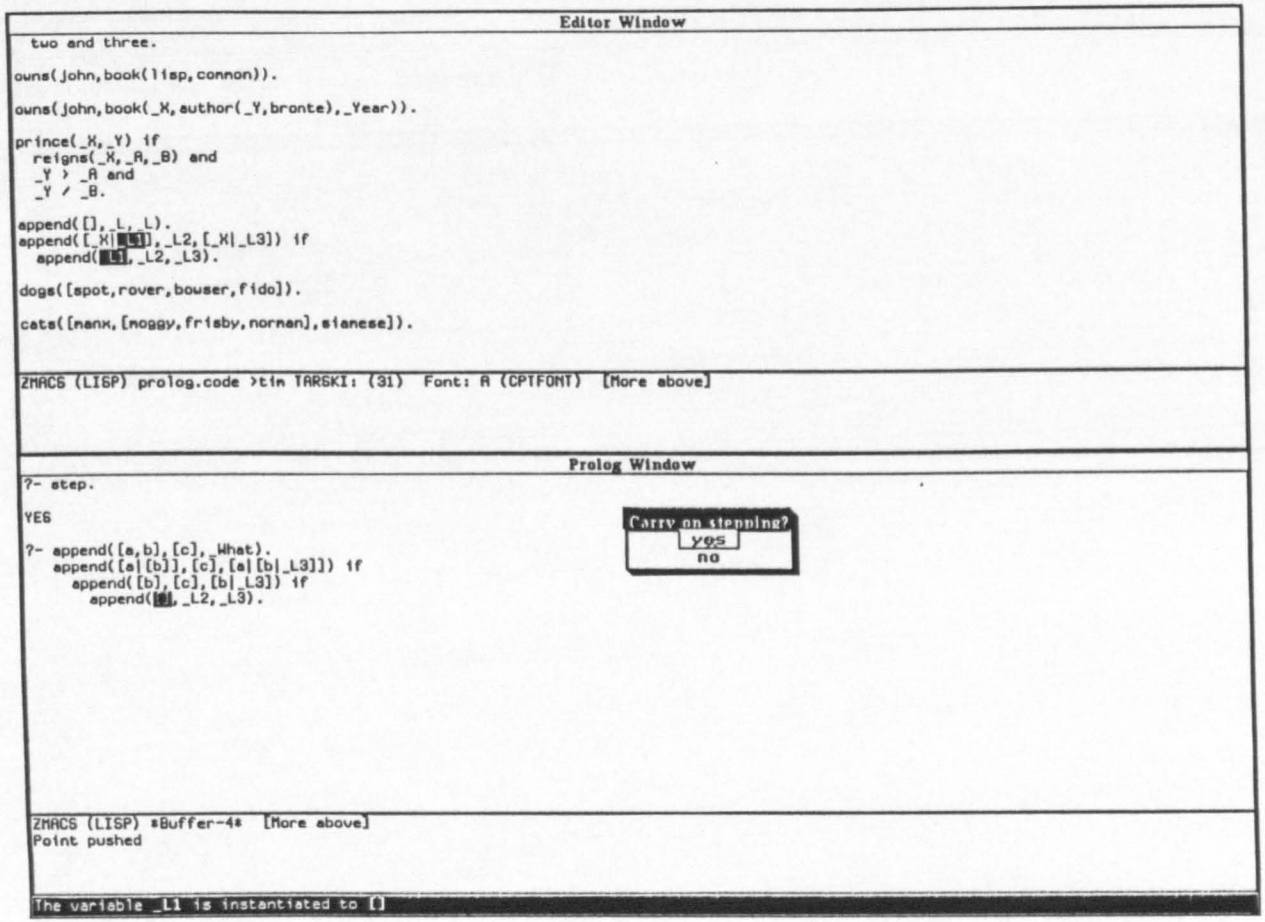

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[moggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append([],_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _L1 is instantiated to []

Figure 6.19

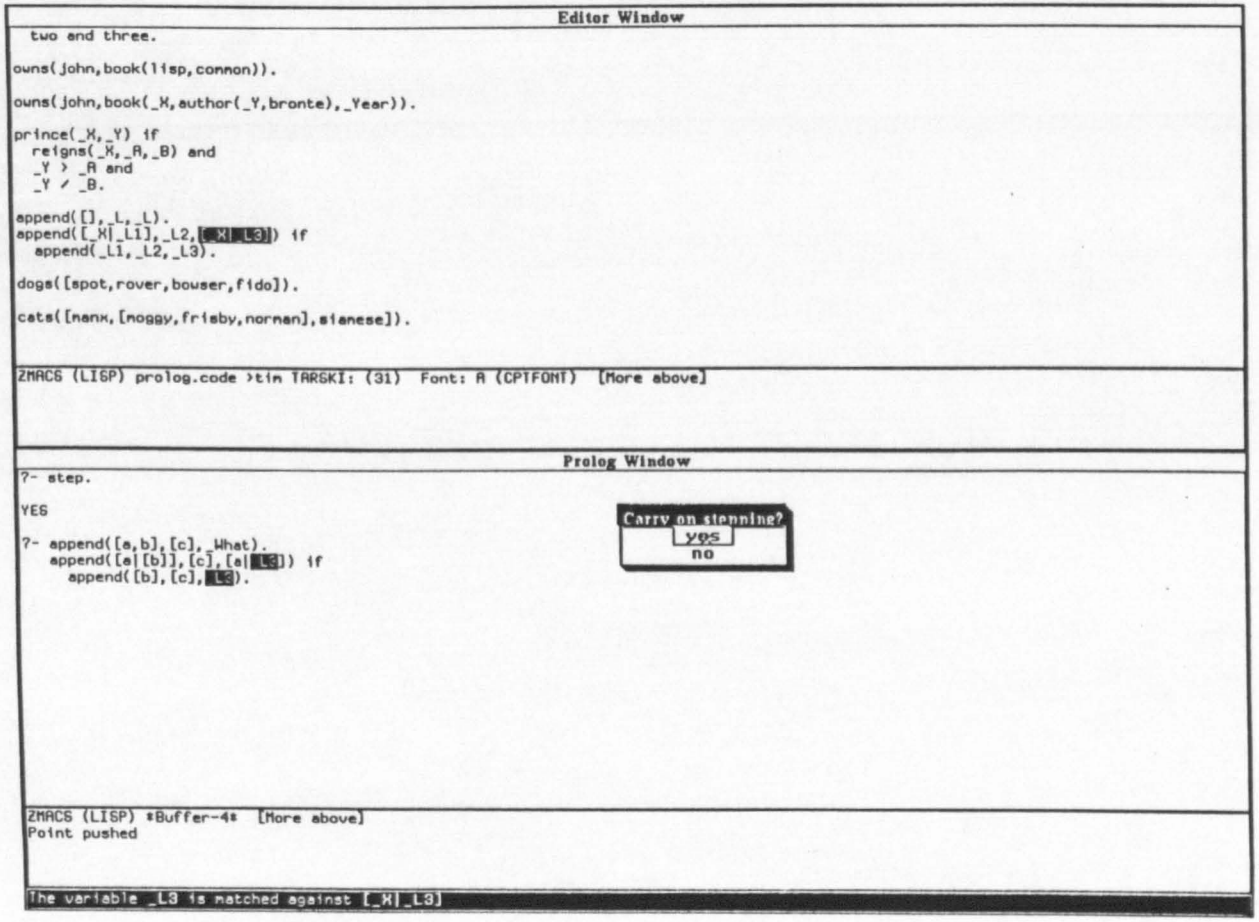

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_L3]) if
append([b],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _L3 is matched against [_X|_L3]

Figure 6.20

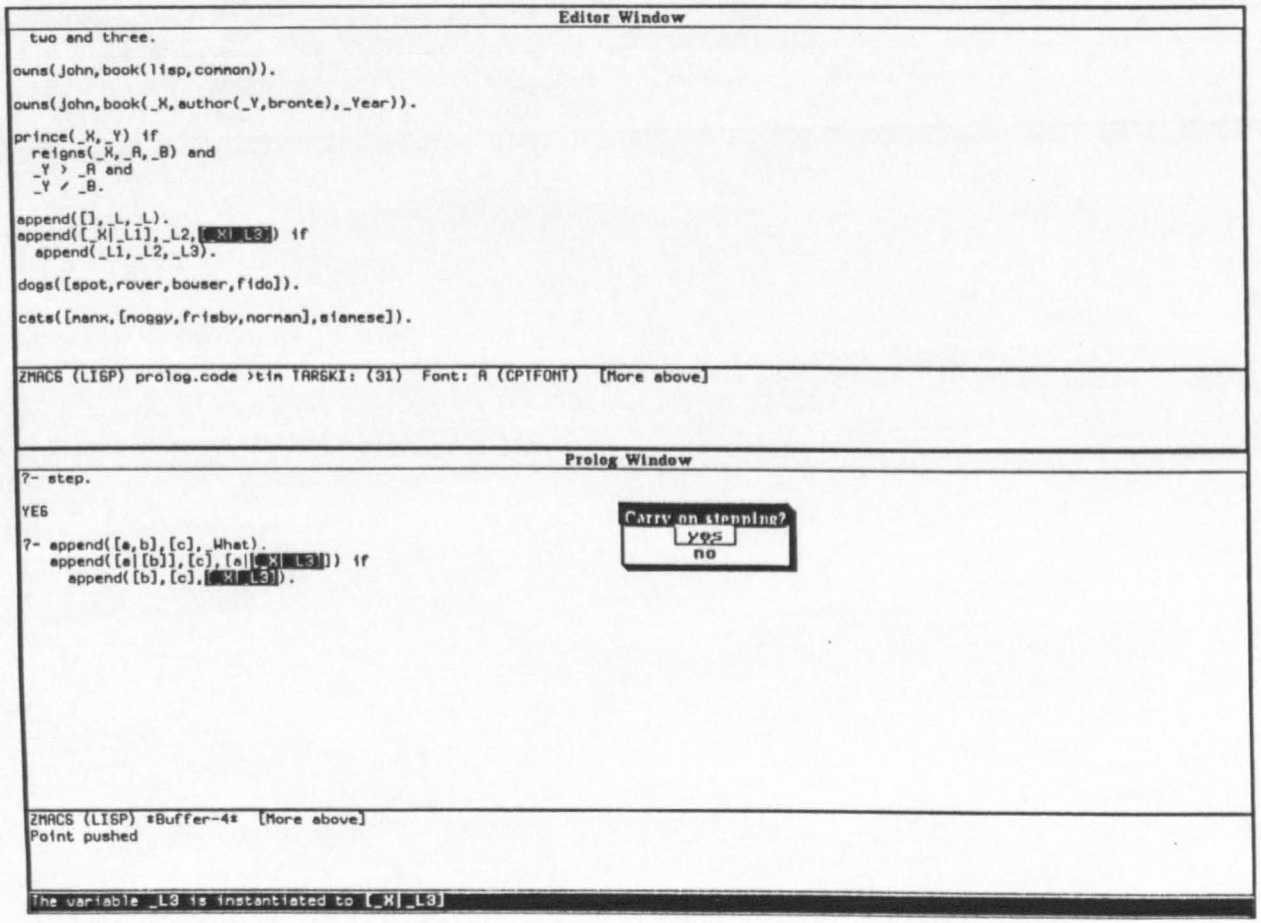

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_X|_L3]]) if
append([b],[c],[_X|_L3]).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _L3 is instantiated to [_X|_L3]

Figure 6.21

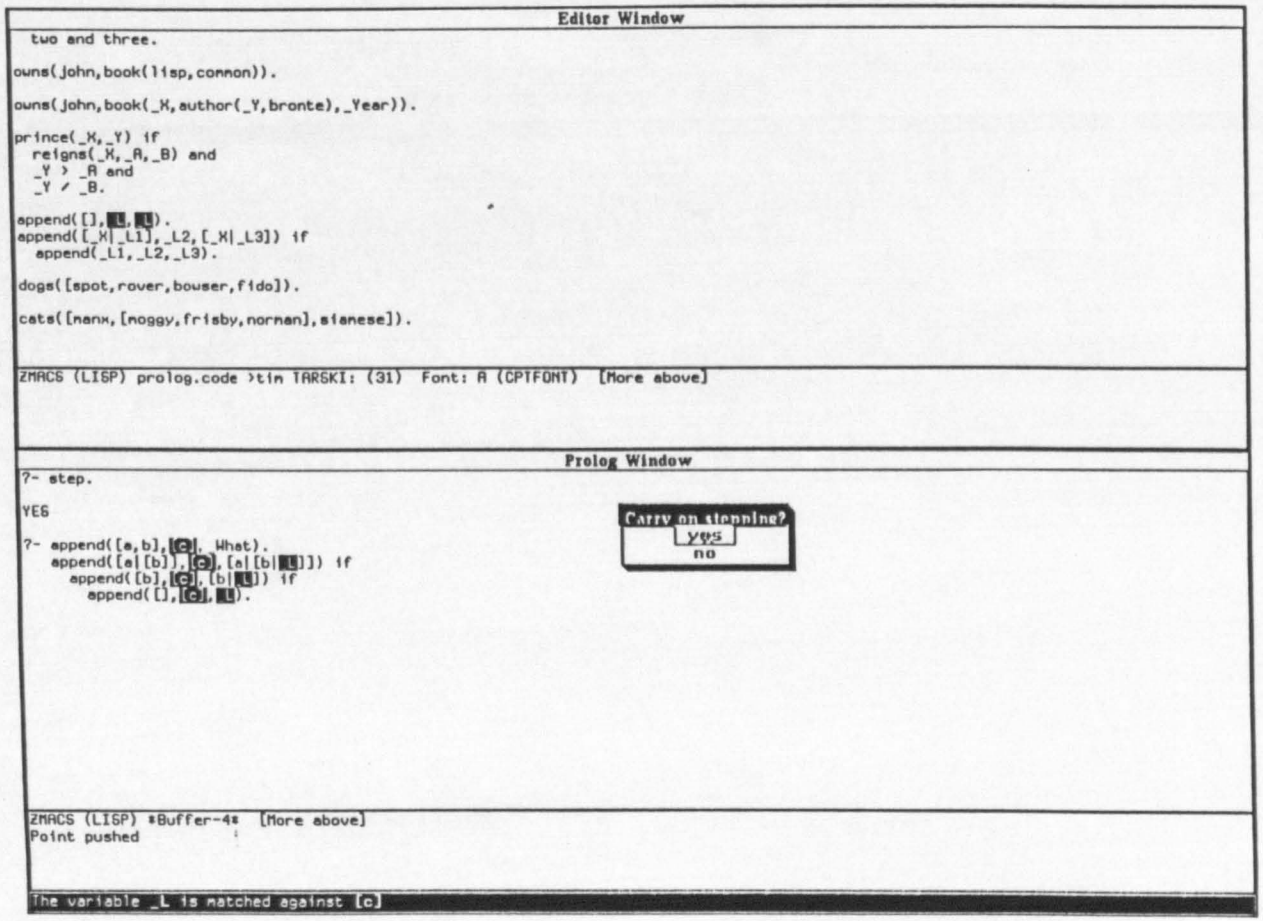

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],L,L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],What).
append([a|[b]],[c],[a|[b|L]]) if
append([b],[c],[b|L]) if
append([],[c],L).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _L is matched against [c]

Figure 6.22

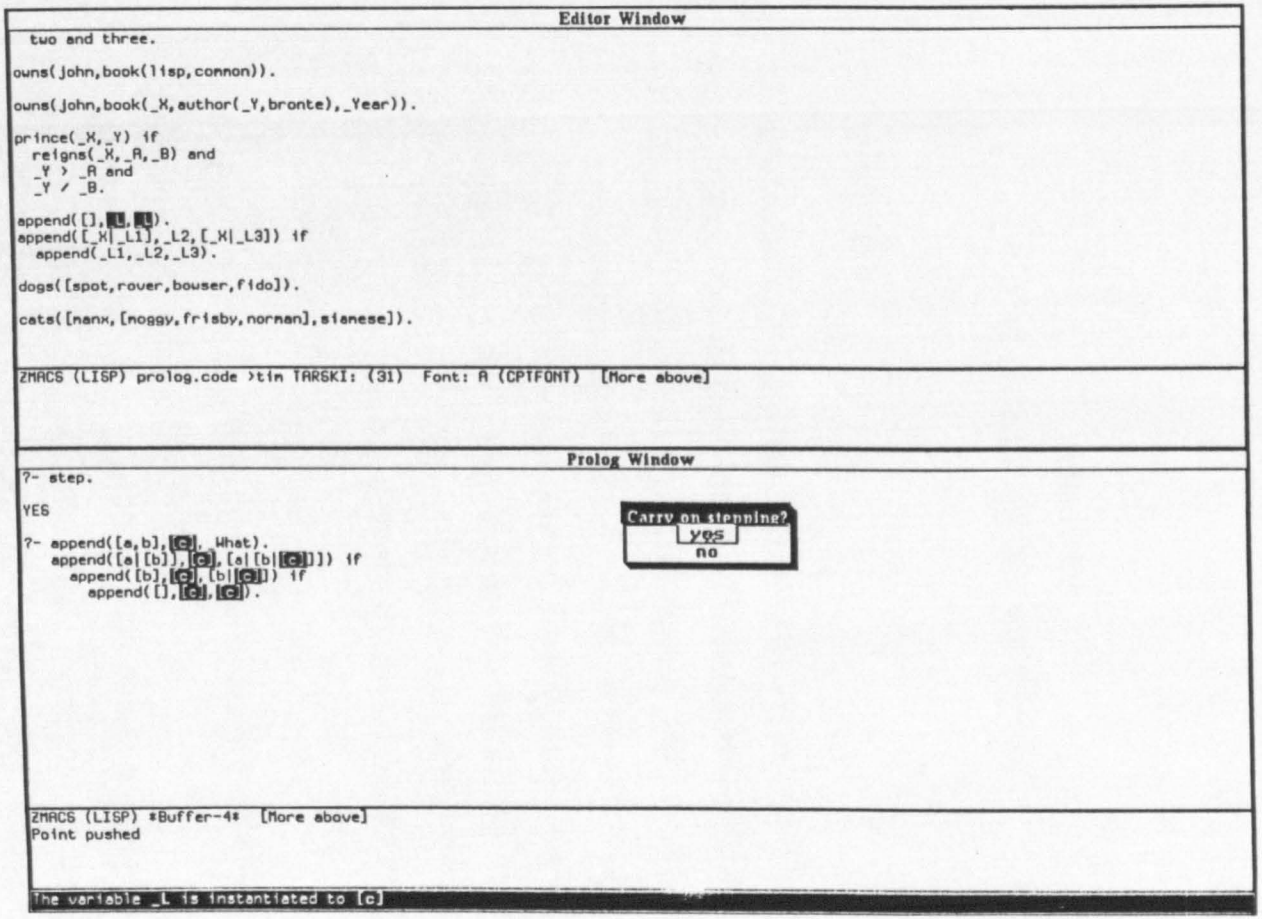

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],What).
append([a|[b]],[c],[a|[b|[c]]]) if
append([b],[c],[b|[c]]) if
append([],[c],[c]).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-4* [More above]
Point pushed
The variable _L is instantiated to [c]

Figure 6.23

# CHAPTER 7

# THE EFFECT OF APT ON NOVICES

## 7.1 Introduction

Chapter two presents the previous research carried out concerning the behaviour of novice programmers, design principles for computing systems and the attempts made to display an animated view of program execution. This provided a base for building a set of design principles to act as guidelines for the designers of animated tracing tools, which are presented in chapter 3. Chapters 4 and 6 discuss the two stages of building an animated tracing system for Prolog based on the amassed design principles. However, this leaves a gap in the design cycle, and that is to determine how the new system and design principles affect the users it has been designed for in the task it is meant to carry out.

In chapter 1 I made the following predictions about the effect of the approach of dynamic tracing tools on novice programmers:

> 1) Students will be able to learn through visual examples what is actually happening in the program. These visual examples are in effect a concrete embodiment of the workings of a program.
>
> 2) Students will be able easily to debug their mistakes by understanding them in terms of the evaluation of the code.
>
> 3) Students will be able to develop their programs more quickly due to the ease of monitoring the surface evaluation of the program.

This chapter presents an empirical experiment looking at the first prediction in order to fill the gap mentioned above. The first prediction is tested rather than either

of the other two for two reasons. Firstly, in order for the second and third predictions to work the first prediction must hold true. This is because if APT does not improve students understanding of program execution neither will it aid them to spot, understand and correct bugs. Secondly, in order to test the other predictions students would have to use APT to debug and develop their own programs. This would require APT to be a very robust system which would have lengthened the development time of APT considerably, and for the purposes of this thesis unneccesarily.

The experiment aims to determine whether APT conveys the required information about the dynamic processes that occur in programs at run-time to novice programmers, by assessing how well subjects understand the action of Prolog programs before and after they have seen animated demonstrations of these programs provided by APT.

Many researchers in the position of evaluating a new programming tool use the method of comparing the relative success of different systems in carrying out the task they were designed for (Roberts, 1983). This method was not used in this case because the aim of this experiment is to evaluate the design principles upon which the system is based rather than the system itself, so comparing it to other similar systems would only accomplish a ranking of the different systems in their success at carrying out different tasks. In addition to this there are a plethora of such systems being churned out by commercial companies, all of which are slightly different in some way, and so to carry out a comparative study would require a long time to acquire the systems, not to mention great expense, and a major long term study needing many subjects for a strong experimental design. For obvious reasons this approach was not feasible, and so the small scale studies presented below were chosen.

As a precursor to this experiment this chapter also discusses two experiments that aim to determine: 1) the ability of novices to explain the meaning of rules in English;

and 2) the misconceptions about the execution of Prolog programs that are held by novice programmers.

## 7.2 The Questionnaire

All the experimental studies use a questionnaire consisting of Prolog programs. As the questionnaires used in the studies are the same I will present a description of the programs that it contains here, rather than repeat it for each experiment. Each program is presented, described, and has been given a number and a mnemonic name for easy reference.

The programs used in this experiment were carefully devised to become progressively more complex in their action at run-time ranging from simple database search to the more complex action of backtracking.

Note that certain of these programs are not used in the 'Explanations' study, and when the results are presented 'N/A', not applicable appears for these programs.

### *7.2.1 Program 1 - 'Warm-up'*

```
likes(mary wine).
likes(mary food).
likes(john food).
likes(john mary).

bothlike(_X) if
   likes(mary _X) &
   PP(_X) &
   likes(john _X).

Q: bothlike(_X).
```

The first program is very short and simple consisting of a simple database search through a small database. This provides the subject with an easy problem to warm up on, getting used to the experimental procedure.

### *7.2.2 Program 2 - 'likes'*

```
has(fred money).
has(joe money).
has(james money).

kisses(jane fred).
kisses(june james).

likes(_X _Y) if
   has(_Y money) &
   PP(_Y) &
   kisses(_X _Y).

Q: likes(_X _Y).
```

'likes' is a similar program to the first with a simple database search through a small database. The difference here is that the query asks for all solutions to be given, which requires the subject to use backtracking to find alternative solutions.

### *7.2.3 Program 3 - 'connected'*

```
origin(BA137 Chicago).
origin(TWA194 Dallas).
origin(PA100 London).
origin(AZ129 London).

destination(TWA194 Paris).
destination(PA100 Rome).
destination(AZ129 Pisa).

stopover(BA137 Washington).
stopover(TWA194 Boston).
stopover(AZ129 Rome).

connected(_F1 _F2) if
   destination(_F1 _X) &
   PP(_F1) &
   origin(_F2 _X).
connected(_F1 _F2) if
   destination(_F1 _X) &
   PP(_X) &
   stopover(_F2 _X).

Q: connected(_F1 _F2).
```

This program is slightly more complex and longer than the previous two. It contains two rules capable of solving the same goal i.e. which two flights have airport connections. The first rule fails to find two connected flights. This requires the subjects to look for alternative solutions to the first goal, with backtracking, before

moving on to the second rule. The second rule succeeds with its second alternative solution.

### *7.2.4 Program 4 - 'has_flu'*

```
kisses(mary john).
kisses(john june).

hasflu(_X) if
   PP(_X) &
   kisses(_Y _X) &
   hasflu(_Y).
hasflu(mary).

Q: hasflu(june).
```

'has_flu' is recursive, with a tail recursive call. This will require the subject to have some knowledge of recursion in order to solve the problem, which goes through two recursive calls before succeeding.

### *7.2.5 Program 5 - 'sisters1'*

```
sisters(_X _Y) if
   female(_X) &
   parents(_X _M _F) &
   parents(_Y _M _F).

female(alice).
female(june).
female(mary).

parents(sue victoria fred).
parents(alice victoria albert).
parents(mary susan edward).
parents(june victoria albert).
parents(jenny jill roy).

Q: sisters(_X _Y).
```

This program is the classic 'sisters' problem which given the rule that X is the sister of Y if X is female and both X and Y have the same parents. This requires subjects to use simple database search to provide all the solutions in the same way as program 2. If they use this method they should get the correct answer, or at least the fact that X can be X's sister using this definition. However if they try to solve the problems semantically with a bottom-up approach, or have an incorrect notion of database search, they will get the incorrect answer.

### *7.2.6 Program 6 - 'sisters2'*

```
sisters(_X _Y) if
   female(_X) &
   parents(_X _M _F) &
   parents(_Y _M _F).

female(alice).
female(june).
female(mary).

parents(sue victoria fred).
parents(alice victoria albert).
parents(mary susan edward).
parents(june victoria albert).
parents(jenny jill roy).

Q: sisters(alice alice).
```

Program 6 is the same as program 5 but with a different query. The query is as follows 'sisters(alice, alice).'. Do subjects realise that this is possible and change their minds or will they still rely on their semantic solution.

### *7.2.7 Program 7 - 'abstract1'*

```
a(_X) if
   b(_X) &
   c(_X).

b(_X) if
   h(_X _Y) &
   PP(_X _Y) &
   i(_Y).
b(_X) if
   h(_Y _X) &
   PP(_Y _X) &
   i(_Y).

h(john mary).
h(jim sue).

i(mary).
i(jim).

c(mary).
c(sue).
c(fred).
c(jane).

Q: a(_X).
```

This is a program from a paper by Coombs and Stell (1986) which has been slightly modified. This program like program 3 contains two rules to solve the same

goal. As in the previous program the first rule fails with all the alternative combinations of data, causing backtracking to the second rule, which succeeds with the second alternative set of data. This program requires subjects to understand both the order of database search and backtracking. Instead of using rules with everyday meanings this program uses letters, and so the novice will have to realise that such programs work in the same way as the others.

### *7.2.8 Program 8 - 'abstract2'*

```
a if
   b &
   loves(John Mary).
a if c.

b if
   d &
   e.
b if f.

c if PP(foo).

d if PP(bar).

e if PP(baz).

e if PP(gort).

f if PP(fez).

loves(John Sue).

Q: a.
```

Program 8 is the last program in the questionnaire. This program consists of a simple database search with failure driven backtracking. However the predicates (heads of the rules/facts) contain no arguments which makes them look strange, and gives them no everyday meaning in their description.

## 7.3 Explanations Experiment

The aim of this experiment is to determine the ability of novice programmers to explain the meaning of different types of rule contained in Prolog programs. It is predicted that subjects will be able to describe programs that can be given an everyday description, but will find it more difficult to explain programs of an abstract nature.

### *7.3.1 Subjects*

Subjects were recruited from the Open University Cognitive Psychology Summer School. All the subjects were novice Prolog programmers, having no previous experience of Prolog other than this course. The extent of their programming knowledge consisted of a 95 page Prolog primer (Eisenstadt, 1986) which teaches the following concepts; facts and queries; the query interpreter; conjunctive queries; rules; pattern matching; database search; the processing of rules by the interpreter including variable instantiation and uninstantiation, and backtracking; a dayschool which provided the subjects with their first hands on experience with Prolog; and lastly a two day Artificial Intelligence project at Summer School. The project requires students to build a simple model of a cognitive process such as the Collins and Quillian model of semantic memory (1969). The project consists of an algorithm specifying the model being built, and a Prolog program embodying that algorithm. Students generally spend around one and a half days developing the program with the continual help of a tutor.

10 subjects from this population were used for this study.

### *7.3.2 Questionnaire*

The questionnaire consisted of the programs described in section 7.2 with the exception of the 'has_flu', and 'sisters2' programs. Included in the questionnaire was an instruction sheet and a model answer. A full listing of this material can be seen in Appendix A, and an example is presented below:

```
likes(mary wine).
likes(mary food).
likes(john food).
likes(john mary).

bothlike(_X) if
   likes(mary _X) &
   PP(_X) &
   likes(john _X).

Q: bothlike(_X).
```

---

Explain what the rule 'bothlike(_X)' means in English, ignoring the print statement 'PP(_X)'

Figure 7.1 An example from the explanation study questionnaire

### *7.3.3 Method*

This experiment was conducted at long distance by post. Each subject was sent a questionnaire, and instructed to read it carefully, taking special note of the example which gives a model of how to answer the questions. Subjects were allowed to answer the questionnaire in their own time. When completed the subjects returned the questionnaire in addressed pre-paid envelopes.

### *7.3.4 Results and Discussion*

The results from this study concerning the explanation of the meaning of specified rules can be used to determine how well the subjects understand the meaning of written Prolog rules.

Table 7.2 shows the percentage of times that subjects described the rules in the programs correctly for each program. The first row, 'correct answer - general', represents the percentage of subjects that correctly described the meaning of the rules in the program in general terms, using variable names. The second row, 'correct answer -specific', represents the percentage of subjects that correctly described the meaning of the rules in the program in terms specific to the data in that particular

program, i.e. using constants. The third row presents the total percentage of subjects who described the rules correctly.

| program | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| correct answer - general % | 69.2 | 69.2 | 88.5 | N/A | 61.5 | N/A | 42.3 | 30.8 |
| correct answer - specific % | 23.1 | 23.1 | 0 | N/A | 7.7 | N/A | 6.7 | 0 |
| **Total %** | 92.3 | 92.3 | 88.5 | N/A | 69.2 | N/A | 50.0 | 30.8 |

Figure 7.2 Percentage correct descriptions of meaning of program

The overall figure for the percentage of correct descriptions, correct descriptions given in terms of the program (specific), and the total are:

| | |
|---|---|
| correct descriptions | 57.7% |
| specific descriptions | 8.5% |
| **Total** | **66.2%** |

Figure 7.3 Total correct descriptions

Figure 7.2 shows that as the programs become more complex in their content subjects find it more and more difficult to explain what the rules in the program mean, especially for programs 7 'abstract1' and 8 'abstract2' which have an abstract nature. This difference between programs with an abstract meaning and those with an everyday meaning is exemplified by programs 3 'connected' and 7 'abstract1'. These programs are interesting because they are essentially similar programs with the rules containing the same number of subgoals and having similar actions at run-time. The only difference is that the rules in program 'connected' can be given an everyday meaning while those in program 'abstract1' cannot. Subjects found it much easier to correctly describe the meaning of program 'connected' (88.5%) than program 'abstract1' (50.0%).

Overall 66.2% of the rules were described correctly by the subjects, which is a fairly low figure bearing in mind that the rules used are short and simple in content. However if this figure is split in programs with an abstract nature (programs 7 'abstract1' and 8 'abstract2') and those with an everyday meaning (programs 1 'warm-up', 2 'likes', 3 'connected' and 5 'sisters1') a large difference becomes apparent. 83.3% of the descriptions of rules contained in programs with everyday meanings were correct, while only 40.4% of descriptions of rules in abstract programs were correct.

The difference between the figures for the correct explanation of concrete/abstract programs of 40.4% and 83.3% suggest that subjects found it more difficult to understand the abstract programs than the programs that had everyday meanings. A statistical test to determine the significance of this difference was not carried out because of the very small sample size.

The above finding also suggests that the subjects should likewise find it more difficult to write programs that are abstract rather than concrete in their definition. If this is the case then it suggests that abstract programs should be avoided when teaching Prolog to novices until they have grasped the fundamentals of Prolog. To test this hypothesis would require a further experiment along the same lines but using a larger sample of concrete and abstract programs which were matched both for run-time and textual complexity.

In summary, novices appear to have little difficulty of understanding the meaning of Prolog programs as long as they have an everyday meaning, which they can use as a framework for building a description. It would seem obvious that in order for novices to solve Prolog queries they must understand the meaning of the rules contained in the program. To check this out it is neccessary to compare the ability of novices to describe the rules in programs with their ability to solve queries to those programs. The next experiment tests this ability to solve queries to Prolog programs and such a comparison is presented at the end of section 7.5.

## 7.4 Misconceptions Experiment

The aim of this experiment is to determine the types of misconception that novices hold concerning the events occurring when Prolog programs are executed. This will allow a categorization of novice misconceptions in terms of the workings of the Prolog interpreter instead of in terms of the bugs made in programming (van Someren, 1984; 1985). A categorization such as the one above will tell us what state of knowledge that the experimental population has concerning their ability to program in Prolog.

### *7.4.1 Subjects*

33 Subjects were used for this experiment, taken from the population of Open University Summer School students described in section 7.2.1. It is possible that the same subjects were used for both this experiment and the explanations experiment. However because the experiment was conducted via the post and the subjects remained anonymous we can not tell if there was any over lap.

### *7.4.2 Questionnaire*

The questionnaire used in this study is the same as that described in section 7.2. It consisted of eight Prolog programs, two model answers, an instruction sheet, and a tutorial on the use of the print statement 'PP'. A full listing of this material can be found in Appendix A. A sample question taken from the questionnaire is shown below:

```
has(fred money).
has(joe money).
has(james money).

kisses(jane fred).
kisses(june james).

likes(_X _Y) if
   has(_Y money) &
   PP(_Y) &
   kisses(_X _Y).

Q: likes(_X _Y).
```

*Note ALL answers are required*
A's:

Figure 7.4 An example from the misconceptions study questionnaire

To assess how the subjects went about the task of answering these questions, print statements 'PP' were inserted into the program. The output from these print statements, giving the values of variables at different points in the program, is meant to provide insight into the execution path used.

For each program the subject is given a query to solve which contained either variables, constants or just consisted of a fact (with no arguments). Sometimes, as in the example above, subjects are required to provide all the solutions, while at other times only one solution is asked for.

### *7.4.3 Method*

This experiment was conducted at long distance by post. The subjects were sent the questionnaire and told to read the instructions carefully and take notice of the model answers. They were allowed to solve the queries in their own time. When completed the subjects returned the questionnaire in addressed pre-paid envelopes.

### *7.4.4 Results*

The results from this study have been analysed in the following way. The different answers to each program in the questionnaire have been grouped together.

Each group of answers has then been analysed to determine the strategy that the subjects used to produce that answer. The results have been analysed by looking at the printout and bindings that the subjects gave as answers. This gives an insight into the strategy used to simulate the execution of the programs in their heads, or in other words it shows the model of program execution that the subjects have.

The correct answers to the programs/queries and the answers that the subjects gave are presented by category of misconception in Appendix D.

The results of the above analysis were then studied to attempt an explanation of how the subjects arrived at each group of solutions given. Explanations were produced using two methods. The first was to alter the action of the Prolog interpreter, and the second was to impose real world constraints onto the program, until a strategy could be mapped onto the solutions. This generated categories of misconception held by the subjects concerning the action of Prolog at execution time.

To make it easier to follow the results I will first provide a description of the categories of misconception which are used to explain the answers that subjects produced. For ease of presentation there will be categories for correct and for answers that cannot be interpreted in terms of a misconception of program execution.

**Correct:** This category represents correct answers.

**Correct - no printing:** Here the correct bindings are given, but no printed output. There is not enough information here to determine whether a misconception concerning program execution has arisen, or not.

**Static match:** This represents an approach where novices attempt to use a rule as a template into which they try and fit the facts contained in the database. This is done in a static manner, where the facts either fit and the rules succeeds, or they do not and the rule fails. No account is taken of partial matches within the rule, and the side effects that can thus be caused. This method can be useful for experts to estimate the action of a progam at a glance but misses out many events which occur at run-time.

**All subgoals must succeed/Static match:** This category of answer can be explained either by 'static match' described above, or by 'all subgoals must succeed' which is described as follows. Unless all the subgoals of a parent goal succeed none of the subgoals can be executed. The misconception here is that all the subgoals of a clause must be unified before any of the subgoals can succeed, rather than each subgoal goal must be unified before it can succeed.

The above two approaches can produce the same answer, but this is not always the case. The difference between 'static match' and 'all subgoals must succeed' is that with the former the user works in a bottom-up fashion, fitting the facts into the rules, ignoring such things as backtracking and correct database search order. With the latter the user works through the program in a top-down manner. This is similar to the Prolog interpreter, but differs in the belief that all of the subgoals of a rule must succeed in order for any one of them to succeed and be executed. This appears to be a misconception concerning unification.

**Real world fallacy:** This is where a novice understands the every day meaning of a Prolog rule but does not understand how the program will work, given a query, at run-time. The novice answers the query by using his/her real world knowledge to interpret the rules, and then fit the facts into this real world knowledge. So the novices are substituting real world knowledge for the Prolog interpreter. This approach will nearly always cause problems, because the way Prolog works very often gives counter-intuitive answers to queries.

**Do not search from the top:** With this misconception novices do not start searching the database from the top each time there is a fresh goal call, but instead carry on from where they are. This will cut out large chunks of the search space, preventing certain solutions, and side effects.

**Only try the first rule:** When there is more than one clause with the same head in the database, the novice will try the first one and if it fails they will stop there and give up. The other rules with the same name are not tried. As with the last misconception this aproach will cut down the search space, and result in no answer where one or more solutions may exist.

**No backtracking:** If in unification a rule fails to match, do not attempt to find an alternative solution by backtracking but move on to the next rule with the same predicate name.

**Miscellaneous:** This category represents all the answers which cannot be explaned in terms of a misconception about program execution. This includes answers of 'no', and where subjects made no attempt to answer the question.

The results are presented in the following manner. For every program in the questionnaire a table is shown giving details of the number of subjects who gave answers for each of the above misconception categories. Below each table an explanation is given describing how each misconception gave rise to the answers that the subjects gave. Where a misconception category does not appear for a particular program the table will show '---', meaning that this category is not applicable to this program.

**Program 1 'Warm-up'**

```
likes(mary wine).
likes(mary food).
likes(john food).
likes(john mary).

bothlike(_X) if
   likes(mary _X) &
   PP(_X) &
   likes(john _X).


Q: bothlike(_X).

A: wine
   food
   _X = food
yes
```

| Category | Number of Subjects | % |
|---|---|---|
| Correct | 16 | 48.5 |
| Correct - no printing | 8 | 24.2 |
| Static match | 3 | 9.1 |
| All subgoals must succeed /static match | 5 | 15.2 |
| Real world fallacy | --- | --- |
| Do not search from top | --- | --- |
| Only try first rule | --- | --- |
| No backtracking | --- | --- |
| Miscellaneous | 1 | 3.0 |

Figure 7.5 Misconceptions - program 1 'warm-up'

*Correct*: These subjects answered the query correctly, with the binding 'food', and the printout 'wine food'. It is assumed that this group of subjects, and other smilar groups in the questionnaire, knew what they were doing in answering the query. This of course may not be the case but without as more detailed investigation it is not possible to find out.

*Correct - no printing*: Subjects got the correct final binding, but did not provide any printed output. This group of subjects may have just not bothered with the print statements, knowing how the program works, or they may not have understood how the print statement worked and ignored them. Either way it is not possible to interpret what was being done by this group and other similar groups in the questionnaire.

*All subgoals must succeed/static match*: Subjects got the correct binding for the query, 'food', but only gave the second output for the print statement, 'food'. There are two possible interpretations for this answer . The first, called *all subgoals must succeed*, is where subjects think that the subgoals of a parent goal only get carried out if they all succeed in matching. In this program the subgoals of the rule 'bothlikes' only succeed with the binding of 'food', not with the binding of 'wine'. So 'food' is printed out but 'wine' is not.

The second interpretation is that subjects are using a bottom-up approach to solving the problem, in other words instead of working through the program in the way that the interpreter would, they attempt to use the rule as a template and match it against the facts in a static manner. Using this approach, called *static match*, 'food' is the only answer that fits the template of the rule, thus getting printed out and being the final binding. However 'wine' does not fit into the template and so will not be printed out.

*Static match*: In this case subjects have provided the correct binding for the query, 'food', but have got the printout in the reverse order, 'food wine'. It appears that the subjects have used the static match approach in the same way as described above, but in addition to this they have gone slightly further. After they have found the solution i.e. food, they have gone back to see if any other database items fit the 'bothlikes' rule. 'Wine' partially fits the rule, and causes the printout of wine.

*Miscellaneous*: This category consists of one subject, who printed out the value of the second argument of 'likes' facts whether its first argument matched the first

argument, given in the 'likes' clause of the 'bothlikes' rule ('mary') or not. It is not clear what this subject was doing in answering this question, but looking at the other answers given by this subject in the questionnaire it seems that this subject did not fully understand how the print statement works in conjunction with the action of the interpreter. It appears that the subject does not understand the concept of unification and database search.

**Program 2 'likes'**

```
has(fred money).
has(joe money).
has(james money).

kisses(jane fred).
kisses(june james).

likes(_X _Y) if
   has(_Y money) &
   PP(_Y) &
   kisses(_X _Y).


Q: likes(_X _Y).

Note ALL answers are required
A's: fred
     _X = jane _Y = fred
     joe
     james
     _X = june _Y = james
no
```

| Category | Number of Subjects | % |
|---|---|---|
| Correct | 23 | 69.7 |
| Correct - no printing | 2 | 6.1 |
| Static match | --- | --- |
| All subgoals must succeed /static match | 4 | 12.1 |
| Real world fallacy | --- | --- |
| Do not search from top | --- | --- |
| Only try first rule | --- | --- |
| No backtracking | --- | --- |
| Miscellaneous | 4 | 12.1 |

Figure 7.6 Misconceptions - program 2 'likes'

*Correct*: Subjects in this category provided the correct answer to the query.

*Correct - no printout*: This group of subjects gave the correct bindings but no printed output.

*All subgoals must succeed/static match*: These subjects got the correct bindings for both solutions but left out the output of 'joe' caused by the print statement. The subjects appear to be using a *static match* or *all subgoals must succeed* method of solving the problem. With the *static match* approach the subjects can fit 'jane fred' and 'june james' into the 'likes' rule, so that all the subgoals succeed. However with 'joe' the template is not filled and 'joe' does not get printed out.

With the all subgoals must succeed method, all the subgoals of the 'likes' rule succeed with the bindings of 'jane fred' and june james', but not with 'joe'. So the correct output is produced for the print statement when the rule succeeds, but the printing is left out when all the subgoals do not succeed.

*Miscellaneous*: This category contains four different answers, each given by one subject.

Only the first solution was given by this subject. This is probably a case where the subject has misread the requirement for all the solutions to be provided, and has only given the first solution.

This subject has given the second solution to the query, and left out the first. Because this subject has got all the correct output from the print statements in the correct order, it is likely that s/he has made a mistake and forgotten to write down the bindings for the first solution.

The subject has given the first solution and the printout of joe. This answer is difficult to interpret. However, this subject had difficulty answering the questionnaire only attempting five of the eight programs, and appeared to have a problem understanding the action of the print statements. One would expect that as the subject has got the first solution, and printed out 'joe' that s/he would go on to get the next solution.

The subject has given the output in this case is 'fred' and 'james'. It looks like this output represents the bindings for the first argument of the query only, ignoring the second one, and also ignoring the printing. This subject ignored the print statements throughout the questionnaire, apart from the last question where the only output is from print statements. The answer does not contain enough information to discern how the subject has attempted to solve this problem, but it appears that s/he has problems with variable binding.

**Program3 'connected'**

```
origin(BA137 Chicago).
origin(TWA194 Dallas).
origin(PA100 London).
origin(AZ129 London).

destination(TWA194 Paris).
destination(PA100 Rome).
destination(AZ129 Pisa).

stopover(BA137 Washington).
stopover(TWA194 Boston).
stopover(AZ129 Rome).

connected(_F1 _F2) if
   destination(_F1 _X) &
   PP(_F1) &
   origin(_F2 _X).
connected(_F1 _F2) if
   destination(_F1 _X) &
   PP(_X) &
   stopover(_F2 _X).


Q: connected(_F1 _F2).

A: TWA194
   PA100
   AZ129
   Paris
   Rome
   _F1 = PA100 _2 = AZ129
yes
```

| Category | Number of Subjects | % |
|---|---|---|
| Correct | 3 | 9.1 |
| Correct - no printing | 3 | 9.1 |
| Static match | 16 | 51.5 |
| All subgoals must succeed /static match | 4 | 12.1 |
| Real world fallacy | 2 | 6.1 |
| Do not search from top | --- | --- |
| Only try first rule | --- | --- |
| No backtracking | --- | --- |
| Miscellaneous | 4 | 12.1 |

Figure 7.7 Misconceptions - program 3 'connected'

*Correct*: These subjects provided the correct answer.

*Correct - no printing*: This group of subjects gave the correct bindings but gave no printed output.

*All subgoals must succeed/static match*: This group of subjects provided the correct bindings for the query, but only gave the last printed output 'Rome'. This answer can be interpreted by either *all subgoals must succeed*, or *static match*. With the *all subgoals must succeed* approach the first 'connected' rule fails and will therefore not produce any output. The subgoals of second 'connected' rule only succeed with the facts 'detination(PA100 Rome)' and 'stopover(AZ129 Rome)', and so only 'Rome' gets printed along with the bindings, '_F1 = PA100', '_F2 = AZ129'.

The alternative interpretation for this answer is that the subjects are using a bottom-up approach or a *static match*. This method uses the 'connected' rules as templates into which the facts are fitted. No facts can be found to fit the first 'connected' rule, which fails. However the facts 'detination(PA100 Rome)' and

'stopover(AZ129 Rome)' fit the second 'connected' rule. This produces the prinout of 'Rome', and the bindings 'PA100 AZ129' for the variables '_F1 _F2'.

*Static match*: The subjects have given the correct bindings for the query, but have produced printing output for all of the facts in the database that could possibly match against the subgoals of the two rules. In other words these subjects have in addition to the correct bindings and printed output, given extra printed output i.e. 'Pisa'. This answer has been achieved by using the 'connected' rules as templates, and fitting the facts into them, using the *static match* method. This method prints out all the first arguments and then all the second arguments of the 'destination facts, thus producing the extra printed output of 'Pisa'. The subjects have not realised that the solution to the query is found before the 'destination(Pisa)' fact is unified against the second 'connected' rule.

*Real world fallacy*: Here the two subjects gave the answer of 'Rome' only. I think that the subjects understood the meaning of the two rules, and could therefore find the correct answer by using their real world knowledge. However they did not know how to work through the program dynamically, checking for any side-effects or bugs. The subjects therefore used their understanding of the meaning of the 'connected' rules to sift through the database and find two flights that are connected.

*Miscellaneous*: This category contains a group of four subjects that gave different answers (see Appendix D) which are so confused that an interpretation is impossible. Some of the answers show a lack of understanding of variable binding. One of the answers 'TWA194' looks like the query has been attempted and the subject has got the first printout, has become confused and given up.

**Program 4 'has_flu'**

```
kisses(mary john).
kisses(john june).

hasflu(_X) if
   PP(_X) &
   kisses(_Y _X) &
   hasflu(_Y).
hasflu(mary).


Q: hasflu(june).

A: june
   john
   mary
yes
```

| Category | Number of Subjects | % |
|---|---|---|
| Correct | 0 | 0 |
| Correct - no printing | 3 | 9.1 |
| Static match | 4 | 12.1 |
| All subgoals must succeed /static match | --- | --- |
| Real world fallacy | 10 | 30.3 |
| Do not search from top | 4 | 12.1 |
| Only try first rule | --- | --- |
| No backtracking | --- | --- |
| Miscellaneous | 12 | 36.4 |

Figure 7.8 Misconceptions - program 4 'has_flu'

*Correct - no printing*: Three subjects gave the correct answer 'yes', but no printing.

*Static match*: Four subjects gave the printed output of 'john' and 'june'. The only explanation for this is the bottom-up *static match* approach. The subjects have fitted the 'kisses' facts into the 'hasflu' rule and this has given the two print statements in

the order stated above, which is the wrong way round. This approach will not print 'mary' because in the 'kisses' facts nobody kisses mary.

*Real world fallacy*: The subjects answered 'no', which suggests that they have seen the clause 'hasflu(mary)', which obviously does not match with 'hasflu(june)' the query, and so it must fail. They have ignored the 'hasflu' rule completely, only attempting to match the the goal to the facts in the database. This answer may have been produced by the subjects real world knowledge which states that if someone has the flu it is obvious because they have symptoms. You do not have to work out whether a person has kissed someone who has the flu to see if they also have the flu. The subjects can see that mary has flu is present in the database, 'has_flu(mary)', but it states nothing about june having flu. So the answer must be 'no', june does not have the flu.

*Do not search from top*: This group of subjects got the first print statement, 'june', and then finished. This suggests that instead of recursing and taking the recursive goal call, 'hasflu(john)', as being a fresh goal call and start matching from the top of the database, the subjects have attempted to match the call 'hasflu(john)' with the fact 'hasflu(mary)' which fails bringing the program to an end.

*Miscellaneous*: Six subjects produced the printing output of 'john', which is the second item that should be printed. This is a difficult answer to interpret, but it appears that these subjects do not understand the way that Prolog searches through the database when it is attempting to unify goals.

A group of three subjects gave the print output 'mary'. This is the last item that is printed out when the program is run. As with the last answer this is difficult to interpret, but again it appears that the subjects have a problem with the way they think the Prolog interpreter works through the database when looking for an item to match against the goal.

Two subjects answered 'mary john', and 'mary june' respectively. I cannot intepret either of these results.

One subject made no attempt at answering the question.

No attempt has been made to explain why subjects produced the answers in the miscellaneous category because to do so would have meant imposing a misconception category onto the answers for the sake of doing so. This would most likely produce a tenuous and misleading category.

**Program 5 'sisters1'**

```
sisters(_X _Y) if
   female(_X) &
   parents(_X _M _F) &
   parents(_Y _M _F).

female(alice).
female(june).
female(mary).

parents(sue victoria fred).
parents(alice victoria albert).
parents(mary susan edward).
parents(june victoria albert).
parents(jenny jill roy).


Q: sisters(_X _Y).

Note ALL answers are required
A's: _X = alice _Y = alice
     _X = alice _Y = june
     _X = june  _Y = alice
     _X = june  _Y = june
     _X = mary  _Y = mary
no
```

| Category | Number of Subjects | % |
|---|---|---|
| Correct | 0 | 0 |
| Correct - no printing | 0 | 0 |
| Static match | 0 | 0 |
| All subgoals must succeed /static match | --- | --- |
| Real world fallacy | 24 | 72.7 |
| Do not search from top | 7 | 21.2 |
| Only try first rule | --- | --- |
| No backtracking | --- | --- |
| Miscellaneous | 2 | 6.1 |

Figure 7.9 Misconceptions - program 5 'sisters1'

*Real world fallacy*: This group of subjects gave the semantically sensible answer to this problem 'Alice June'. Given the facts and the real world knowledge that people possess this is the obvious answer, that Alice and June are sisters. There is another interpretation of this answer, which is produced with the *dont search from the top* strategy. When the first two subgoals of the 'sisters' rule have been unified and the third subgoal is being matched, then instead of starting the search at the top of the database because of the fresh invocation of the second 'parent' subgoal, the subjects may have carried on from where the first 'parent' subgoal matched. This strategy would prevent all the answers apart from 'alice june' and 'june alice'. However because subjects using this method should have produced both of these answers, it is more likely that they used real world knowledge to get the answer.

One subject gave the following answer; 'alice june, alice alice, june june'. The second subject in addition gave 'mary mary'. Both these subjects have some notion of how Prolog works through the database at run-time, but have gone for the semantically sensible answer first. The subjects have used real world knowledge to get the first solution 'alice june', but have then used some other method to produce

the other solutions. It is not clear what strategy these subjects used in getting the rest of the answers. Both subjects missed the 'june alice' solution, and the first subject missed the 'mary mary' solution, but as s/he got the 'alice alice' and 'june june' answers this may just be a slip. However this may again be due to real world knowledge, because if you know that alice is the sister of june then you also know that june is the sister of alice. This makes the latter piece of information redundant in real life.

*Do not search from the top*:Seven subjects gave the answer 'alice june', 'june alice' which has been described in the above category.

*Miscellaneous*: One subject gave two solutions; 'alice alice' and 'alice june'. This is the correct first two answers, but no more are given. It is possible that the subject got confused at this stage, because a lot of information, pertaining to where in the database the search has got to, has to be remembered. If the subject did get lost in this way then s/he may have given up. Altenatively s/he may have thought that there were no more answers.

One subject made no attempt to answer the question.

**Program 6 'sisters2'**

```
sisters(_X _Y) if
   female(_X) &
   parents(_X _M _F) &
   parents(_Y _M _F).

female(alice).
female(june).
female(mary).

parents(sue victoria fred).
parents(alice victoria albert).
parents(mary susan edward).
parents(june victoria albert).
parents(jenny jill roy).


Q: sisters(alice alice).

A:yes
```

| Category | Number of Subjects | % |
|---|---|---|
| Correct | 20 | 60.6 |
| Correct - no printing | --- | --- |
| Static match | --- | --- |
| All subgoals must succeed /static match | --- | --- |
| Real world fallacy | --- | --- |
| Do not search from top | --- | --- |
| Only try first rule | --- | --- |
| No backtracking | --- | --- |
| Miscellaneous | 13 | 39.4 |

Figure 7.10 Misconceptions - program 6 'sisters2'

This program is the same as program 5, but instead of a query with two variables asking for all solutions, this program asks the following query 'sisters(alice,alice)'. The interesting point here is not how many people gave the answer 'yes' and 'no', but how many subjects said 'yes' who did not give the 'alice alice' answer to the last program.

20 subjects (60.4%) gave the yes answer, whereas only 3 (9.1%) gave the 'alice alice' answer last time. 13 subjects (39.4%) who missed the 'alice alice' answer previously still gave a 'no' answer to this query. Around half of the subjects realised that this answer was possible, given the insight of the information in the query, but just under half still did not believe this solution to be possible. This means that these people are either stuck because of their real world knowledge; or they have a problem with the way Prolog searches the database.

**Program 7 'abstract1'**

```
a(_X) if
   b(_X) &
   c(_X).

b(_X) if
   h(_X _Y) &
   PP(_X _Y) &
   i(_Y).
b(_X) if
   h(_Y _X) &
   PP(_Y _X) &
   i(_Y).
h(john mary).
h(jim sue).

i(mary).
i(jim).

c(mary).
c(sue).
c(fred).
c(jane).


Q: a(_X).

A: john mary
   jim  sue
   john mary
   jim  sue
   _X = sue
yes
```

| Category | Number of Subjects | % |
|---|---|---|
| Correct | 10 | 30.3 |
| Correct - no printing | 3 | 9.1 |
| Static match | --- | --- |
| All subgoals must succeed /static match | 1 | 3.0 |
| Real world fallacy | --- | --- |
| Do not search from top | --- | --- |
| Only try first rule | 4 | 12.1 |
| No backtracking | 1 | 3.0 |
| Miscellaneous | 14 | 42.4 |

Figure 7.11 Misconceptions - program 7 'abstract1'

*Correct*: Ten subjects got the correct answer, including the printing.

*Correct - no printing*: Three subjects got the correct binding to the query, but gave no printed output.

*All subgoals must succeed/static match*: One subject gave the correct binding of 'sue', and the printing 'jim sue'. This subject has used the*all subgoals must succeed* approach or the *static match* . With the former the first 'b' rule fails and will there fore not produce any output. The subgoals of the second 'b' rule only succeed with the bindings from the unification with 'h(jim sue)'. This gives the answer '_X = sue' and the printout 'jim sue'.

With the *static match* approach the bindings from 'h(jim sue)' are the only values that fit into the 'b' rules, given the values contained in the 'i' facts. This gives the same answer as above: '_X = sue' and 'jim sue'.

*Only try the first rule*: These subjects gave no final binding but got the following printed output, 'john mary, jim sue', which is the complete printout from either the first or second rule. However, because no solution is given it is unlikely that the

printout is from the second 'b' rule. This suggests that these subjects have tried the first rule, giving the printout, which has failed and because it has failed they have stopped there.

*No backtracking within rules*: One subject gave the correct binding 'sue', with the printout 'john mary' and 'jim sue'. It is likely that the printout 'john mary' has been produced by the first 'b' rule, and because this rule has failed the subject has moved on and attempted the second 'b' rule giving the printout 'jim sue' and the solution '_X = sue'. If this interpretation is correct then the subject has an error in the way s/he conceptualizes unification in Prolog.

*Miscellaneous*: Four subjects got the wrong final binding of 'mary', but got the correct printing output. The only interpretation for this answer is that the subjects managed to follow the program through but got confused towards the end and chose the wrong binding to write down. This is probably because there is a lot of information to be thought about in the program at run-time and it is easy to forget where you are. This however constitutes another problem that novices have, which is remembering the dynamic actions that have taken place at run-time.

Two subjects gave 'mary' as the solution on its own. Two others gave the same solution with a printout of 'john mary'. One subject gave 'john' as the solution with a printout of 'john mary'. Lastly one subject just gave the printout of 'john mary, jim sue, john mary, jim sue' but no solution. This subject may have just got lost and given up.

Three subjects made no attempt to answer the question.

One subject answered 'no', with no printing.

**Program 8 'abstract2'**

```
a if
   b &
   loves(John Mary).
a if c.

b if
   d &
   e.
b if f.

c if PP(foo).

d if PP(bar).

e if PP(baz).

e if PP(gort).

f if PP(fez).

loves(John Sue).


Q: a.

A: bar
   baz
   gort
   fez
   foo
yes
```

| Category | Number of Subjects | % |
|---|---|---|
| Correct | 0 | 0 |
| Correct - no printing | --- | --- |
| Static match | --- | --- |
| All subgoals must succeed /static match | 8 | 24.2 |
| Real world fallacy | --- | --- |
| Do not search from top | --- | --- |
| Only try first rule | --- | --- |
| No backtracking | 3 | 9.1 |
| Miscellaneous | 22 | 66.7 |

Figure 7.12 Misconceptions - program 8 'abstract2'

*All subgoals must succeed/static match*: This group of subjects gave the answer 'foo'. This is the last item that is printed, and corresponds to the success of the second 'a' rule. This can be interpreted in several ways. Firstly it may be a case of *all subgoals must succeed*, where if any of a parent goal's subgoals do not succeed then none of them get executed. So because the first 'a' rule fails due to its last subgoal, the previous subgoal never gets executed and no printout is produced. The second 'a' rule only has one subgoal which succeeds giving 'foo' as its output.

This answer has another interpretation might be that subjects immediatly spot that the first 'a' rule will fail on its last subgoal, so they ignore this rule and move to the second 'a' rule. The second 'a' rule only has one subgoal which succeeds. This is using a *static match* approach, using the rule as a template and matching it against the database entries.

*No bactracking*: This group of subjects produced the following output 'bar baz foo' (two subjects), 'bar baz fez foo' (one subject). This output has been caused by a misconception of the action of the Prolog interpreter when goals fail. In the first answer no bactracking has been carried out within the first 'a' rule, which means that

the second 'b' rule has not been tried. Instead the subject moves on to the second 'a' rule. Also no backtracking has occured within the first 'b' rule. In the second answer both 'a' and 'b' rules have been attempted but no backtracking has been tried within the second 'b' rule.

*Miscellaneous*: Eleven subjects gave the program output as 'foo bar baz gort fez'. This is produced if the 'a' rules are executed in the reverse order. It is also the order in which the items appear in the program code. This suggests that subjects have written the output down in the same order as it appears in the program, or that they have executed the simplest 'a' rule first and the more complex one afterwards. Whichever method the subjects have used, they do not understand how the Prolog interpreter would search through the database to unify the query and subsequent goals.

Two subjects gave the output 'foo fez'. This is produced by the success of the second 'a' and 'b' rules. It is not clear why subjects would only use the second 'a' and 'b' rules, although they are simpler than the first in both cases. It is possible that because the first 'a' rule fails on its last subgoal that they have ignored it. But if this was the case then you would expect both 'b' rules to be ignored. However that they have done so, shows that the subjects have a misconception of how Prolog searches the database in an attempt to unify the goal.

Two subjects gave the following output to the question 'bar baz bar gort foo fez'. This is too complex an answer to be interpreted, but it is clear that these subjects have some confusion about the action of backtracking upon the failure of a goal.

Three subjects did not attempt to answer this question.

Four subjects gave the answer 'no', with no printout.

These answers have been placed in the miscellaneous category because it was felt that no explanation could be found that was strong enough to provide a misconception

category. Suggestions have been put forward concerning what subjects may have done, but they are too tenuous to form a category, and to do so may mislead the reader.

### *7.4.5 Discussion*

The answers that the subjects provided to the questionnaire have pointed out some possible misconceptions held concerning the action of the Prolog interpreter on fairly simple programs at run-time. These misconceptions are summarized in the table below, which shows the programs where they arose.

| **Category** | **Program** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Static match | 1 | | 3 | 4 | 5 | | | |
| All subgoals must succeed/static match | 1 | 2 | 3 | | | | 7 | 8 |
| Real world fallacy | | | 3 | 4 | 5 | | | |
| Do not search from the top | | | | 4 | | | | |
| Only try the first rule | | | | | | | 7 | |
| No backtracking | | | | | | | 7 | 8 |

Figure 7.13 Table of misconceptions concerning program execution

The misconceptions in the above table show that novice programmers have problems with program execution in three areas. All *subgoals must succeed* and *static match* are due to an inaccurate model of what happens in unification. *Do not search from the top, only try the first rule*, and *no backtracking* show a problem with the control structure of the Prolog interpreter. Lastly *real world fallacy* demonstrates that if the program has a real world meaning, then the subject will impose this meaning on the program to produce a solution, rather than follow the mechanism of the interpreter.

Besides the misconceptions described by the categories in Figure 7.13 the results contain other misconceptions which have no clear interpretation. These misconceptions are contained in the *miscellaneous* category (which also includes subjects who made no attempt to answer the question). Explanations have been suggested concerning how these answers may have been produced, but it is felt that they are too loose to provide a useful misconception category. However, the answers contained in the miscellaneous category show that the subjects do not understand many of the concepts inherent to the execution of Prolog programs. The concepts concerned are unification, database search, variable binding, and backtracking .

The design of the questionnaire for this type of long distance study is probably as good as can be expected. One point where the design fell down was the reliance on the printed output given by subjects as an indicator of the strategy that subjects use in attempting to solve queries. This approach has given some insight into what might be happening when the subjects have solved these questions, but a small number of subjects provided no printed output thus defying analysis. Even where subjects did provide output, more information is required in many cases to be sure of the strategy used in answering the question. This extra information would be provided by a verbal protocol of subjects solving these problems, leaving no doubt as to the strategy used and the misconceptions held about the action of the Prolog interpreter. Unfortunately verbal protocols were not a practical method for a long distance study such as this.

## 7.5 Explaining vs Solving Prolog Programs

Section 7.3.4 presents the results from a study into the ability of novice programmers to explain the meaning of Prolog rules. Section 7.4.4 presents the results from a study into the ability of novices to solve Prolog programs. This section compares these results to determine how the ability to explain Prolog rules affects the ability to solve queries to those programs. It was predicted in section 7.3.4 that in

order for subjects to be able to solve Prolog programs they must understand the meaning of the content of that program. So one would expect the results to show a correlation between the ability of subjects to explain rules in a program and the ability to solve those programs.

Table 7.14 shows the figures for the percentage correct answers and explanations. The correct answers include those with and without printing, and the correct explanations include specific and general explanations. Programs which are not applicable are denoted by '---'.

| program | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| % correct explanation | 92.3 | 92.3 | 88.5 | --- | 69.2 | --- | 50.0 | 30.8 |
| % correct answer | 72.7 | 76.8 | 18.2 | --- | 0 | --- | 39.4 | 0 |

Figure 7.14 Comparison of correct explanations and correct answers

The table shows a steadily dropping figure for the number of subjects to correctly explain the meaning of the rules in each program. This has previously been explained by the fact that the programs in the questionnaire steadily become more difficult. The questionnaire starts with programs 1 and 2 ('warm-up' and 'likes') that have an everyday meaning together with a simple run-time action. Programs 3 and 5 ('connected' and 'has_flu') still have an everyday meaning but the run-time action becomes more complex. Finally programs 7 and 8 ('abstract1' and 'abstract2') have an abstract meaning, coupled with a complex run-time action.

A Spearmans $\rho$ test (for tied ranks) was applied to the data in Fig. 7.14 to test for any correlation between the ability to explain the meaning of rules in Prolog programs and the ability to solve those programs. The test gave a value of $\rho = 0.65$. This is not significant at $p = 0.05$ level which requires a value equal or greater than 0.829.

However the reader must bear in mind the fact that for small sample sizes (samples under 10) the correlation coefficient required to reject the null hypothesis is unusually high. For example, as quted above, for a sample size of 6 the required

value of $\rho$ is 0.829 (at the $p = 0.05$ level), while for a sample size of 10 the required value of $\rho$ is 0.564 (at the $p = 0.05$ level) which is well below the value generated by the data in Fig. 7.14.

It is therefore felt that because the correlation coefficient required to reject the null hypothesis for small sample sizes is so high the Spearmans test may be misleading for the data shown above. So bearing the values shown above in mind, and with the reservation that only a more extensive experiment will determine the correct interpretation of the results, the data in Fig. 7.14 has been analysed by eye and given the following interpretation.

It was concluded that if a program had an everyday meaning then the subject could explain the rules in a program, but if no everyday meaning could be found subjects could not explain the program. Taking the drop in ability to explain the rules of the programs into account, one would expect the number of subjects to answer the questions correctly to drop in a similar way. One would also predict from this that if a program had an everyday meaning then subjects would find it easier to solve that if the program was abstract. However, the table shows that this is not true and that being able to explain a program does not mean that the subject can also solve it.

The first two programs, 1 'warm-up' and 2 'likes', have high figures for the correct explanation of rules which result in a high figure for correct solutions. Also program 7 'abstract1' has a relatively high figure for correct answers reflecting the figure for correct explanations which was expected. However the other programs do not reflect this pattern between correct explanation and correct answer. Programs 3 and 5 ('connected' and 'sisters1') have a high explanation figure but a very low correct answer figure. While program 7 ('abstract1') has a relatively high correct answer figure which reflects its reasonably high explanation figure. Finally program 8 ('abstract2') has no correct explanations and a low explanations figure.

How can this be explained considering the predictions that were made? The low figure of correct answers for Program 3 ('connected') is due to the fact that a large number of subjects (51.5%) made a small error. In addition to the correct output for the program they gave one extra printout (see Appendix D). This is a minor error, and if this group were included in the correct answers the percentage figure would rise to 69.7% which is in line with the prediction.

The low correct answers figure for program 5 ('sisters1') is due to an entirely different cause. 72.7% of subjects produced an answer based on their real world knowledge about sisters, rather than the Prolog interpreter. The correct answers to the previous programs that had an everyday meaning coincidentally matched the answers that subjects could get using their real world knowledge. Also because these programs were relatively simple most subjects did not need to fall back on their real world knowledge in order to solve the program. In the case of program 5 ('sisters1') the answer from the interpreter and the real world answer do not match up, and because the program is fairly complex subjects have fallen back on their real world knowledge in order to provide an answer.

The explanation of a zero correct answers figure for program 8 ('abstract2') is not so simple. It may be due to the abstract nature of the program combined with its complex run-time action. However program 7 ('abstract1') has an abstract meaning and a complex run-time action, but 39.4% of subjects managed to solve it correctly. Program 8 does have an unusual structure in that the predicates do not have any arguments, which the subjects had only seen before in the second model answer provided in the questionnaire. In addition to this the factor of fatigue may have crept in. Program 8 is the last program in the questionnaire, which on average takes about one hour to complete.

In conclusion (bearing in mind that the analysis has been carried out by eye rather than statistical test) it appears that if a program has an everyday meaning which the novice can understand, it does not follow that s/he will be able to solve it. If the

novice is not sure how to go about solving the program, then s/he will fall back on his/her real world knowledge to provide an answer. This can be fatal because the answers given by the Prolog interpreter are not compatible with real world knowledge.

## 7.6 APT Experiment

The aim of this experiment is to see whether APT has any affect on the way that novice programmers think about the way Prolog programs work at run-time. This is tested by measuring the ability of novices to solve queries to Prolog programs. In the study, subjects were either allowed to see an animated trace showing the run-time action of Prolog programs, or just the solution to the programs. The subjects who saw the animated demonstration are called the 'APT' group, while the subjects who only saw the solutions are called the 'control' group.

The study consists of two questionnaires and a demonstration. The first questionnaire tests the ability of novice programmers to solve queries to Prolog programs. This is followed by a demonstration, using APT, showing the subjects the process of execution of the programs and queries contained in the first questionnaire. The second questionnaire again consists of a set of Prolog programs and queries which the subjects must solve.

### *7.6.1 Subjects*

Six subjects were used in the 'APT' group. These subjects were drawn from the same population used in the previous experiment, which is described in section 7.3.1. None of these subjects took part in either the explanations or the misconceptions experiment. Six subjects were used in the control group. These subjects are members of the Open University staff, all of whom are novice programmers. The reason that the control group was drawn from a different population was due to the unavailability of subjects from the population used for the experimental group. The

programming experience of the control group consisted of reading the same 95 page Prolog primer that the 'APT' group used. So the programming experience of both groups was matched as far as possible.

### *7.6.2 Method*

The method consists of three parts, two questionnaires on solving Prolog programs split by either the animated demonstrations for the 'APT' group, or the solution to the programs for the control group. The details of the experiment are described below.

Both groups of subjects were given the first questionnaire (Appendix A) which consists of 8 Prolog programs each with a query. The questionnaire also contains a tutorial on the use of the print statement to remind the subjects how the print statement works, and for reference by the subject during the experiment. The subjects were asked to answer these Prolog queries as best they could, and in there own time, writing down the output of the print statements and the final bindings of the varaibles given in the query.

The 'APT' group were then given a three page introduction to the APT system (Appendix C) before being shown a demonstration, using APT, of the way Prolog would solve each of the 8 programs in questionnaire 1. Only one demonstration of each program was allowed due to the considerable length of the experiment. The subjects were allowed to control their movement through APT demonstrations and thus went at a pace that suited them. They were not allowed to ask the experimenter questions concerning the information given in the demonstrations.

The 'control' group were allowed to run the programs from the first questionaire, so that they could see the solution to each program. The subjects could see the source code of each program as they ran it, allowing both source code and solution to be viewed simultaneously. However they were not allowed to trace the programs at any time. The subjects were allowed to carry out the experiment in their own time, and

were not allowed to ask the experimenter questions concerning the solutions to the programs which they ran.

After the experimental conditions a second questionnaire (see Appendix B) was given to each group of subjects. This questionnaire contains programs that are isomorphic in their action at run-time to the programs in the first questionnaire. The subjects were instructed to answer this questionnaire in the same manner as the first one.

### *7.6.4 Questionnaires*

The first questionnaire is the same as the one used in the main study into the misconceptions held by novice programmers concerning those programming concepts that occur at run-time. This is described in section 7.2 and can be seen in detail in Appendix A. The second questionnaire (see Appendix B) consists of programs and queries that are isomorphs of the programs in the first questionnaires. This is to prevent the subjects from remembering their answers to the first questionnaire, or from the APT demonstratiom of these programs, when answering the second questionnaire. Isomorphs were used so that the complexity and type of programs remain the same as for the programs used in the first test.

### *7.6.5 APT Demonstration*

The demonstration of program execution shown to the subjects in between the two questionnaires was provided by APT, which is described in detail in chapter 5. A view of program execution for all of the programs in the first questionnaire was presented to each subject, who was allowed to step through the program at his/her own pace, but without any further instruction than that provided by the APT introduction.

### *7.6.6 Results*

Both questionnaires were scored to determine how well the subjects answered the questions, and this is taken to be a measure of how well the subjects have understood the action of the Prolog interpreter at run-time on the given programs. The difference between the scores achieved on the questionnaires should thus be an indication of the affect of the APT demonstrations on the ability of the novice programmer to understand the action of the Prolog interpreter at run-time. In other words do the principles embodied in APT improve the conceptual model held by novices of the dynamic processes that occur when Prolog programs are executed.

The following method has been used to score the answers to the questionnaires in order to take account of all the aspects of the program output, and subjects answers.

One point is awarded for:

A) any correct item subjects wrote down that should be printed out by a print statement.

B) any two items from print statements that occur in the correct order.

C) any correct final binding for the variable/s given in the query.

D) any two sets of bindings that occur in the correct order, where all solutions are asked for.

In addition to this a point is deducted for any item that the subject writes down that should not be in the program output.

The marks that each subject scored on the two questionnaires can be seen in detail in Appendix E. The total marks that each group of subjects scored on each questionaire is shown in figures 7.15 and 7.16. The maximum score on each questionnaire is 67.

| 'APT' group subjects | RP | RM | VC | TJ | PP | DP |
|---|---|---|---|---|---|---|
| Pre-APT Questionnaire | 12 | 41 | 9 | 9 | 41 | 31 |
| Post-APT Questionnaire | 13 | 46 | 28 | 17 | 44 | 56 |

Figure 7.15 'APT' group scores

| Control group subjects | LC | MD | NP | TG | DM | CR |
|---|---|---|---|---|---|---|
| Pre-solution Questionnaire | 16 | 18 | 13 | 17 | 15 | 9 |
| Post-solution Questionnaire | 16 | 21 | 12 | 15 | 11 | 12 |

Figure 7.16 Control group scores

The results for the 'APT' group show that all the subjects improved their scores on the questionnaire after they had seen the demonstrations of program execution provided by APT. To determine whether the difference in scores are significant a t-test has been applied to the mean scores of each questionnaire. The assumptions underlying the t-test, i.e. that the population distribution is normal and of the same variance, are valid. The difference between the mean scores on the two conditions is significant, $t = 2.43$ $p < 0.05$, one tailed.

The results for the control group show no significant difference, at the $p < 0.05$ or $p < 0.1$ one tailed, between the mean scores of the questionnaires ($t = 0.146$).

### *7.6.7 Discussion*

The results show that there is a statistically significant improvement in the ability of novices to solve queries to Prolog programs after seeing animated demonstration of program execution. The control group shows that this improvement is unlikely to be due to the effects of other variables. However one or two possible problems must be raised. It is possible that the results from the experimental group might have been effected positively (increased their scores) by them learning how to use APT. However it is also possible that the extra information subjects had to remember in learning to use APT may have effected the results negatively (decreased their scores).

One would hope that these two effects cancel each other out, but in any case it is very difficult to determine how these problems have effected the subjects results.

Some subjects in the 'APT' group showed a large improvement, while others only showed a minor improvement.

RM and PP showed only a small improvement in their scores. This may be due to the fact that their original scores were quite high compared with the other subjects. It would therefore be relatively more difficult for RM and PP to increase their score by a large amount than for the other subjects. Another reason why these two subjects only improved by a small amount may be because APT only helped with certain aspects of Prolog execution leaving the subjects unclear about others.

RP showed only a very tiny improvement in her score. It should be noted that RP stated prior to the experiment that she was having some difficulty with learning Prolog. This may point to the fact that APT might not help certain categories of user, suggesting that APT users need to have some basic knowledge concerning Prolog before APT will be of any use in communicating information about Prolog execution.

The small improvement by some subjects may also be explained by the fact that the subjects were only allowed to see each program demonstrated once, and all eight programs were demonstrated one after the other. This means that the subjects were required to take in a lot of information from only one demonstration of the execution of a program. This is a difficult task for novices as they only have a sketchy conceptual framework on which to hang this new information on, which is in fact the reason why this system has been built. One would expect a significant improvement in subjects scores if they were allowed to see further demonstrations. Alternatively this could mean that either APT only helps certain groups of user, or for certain aspects of Prolog execution.

Another factor which may cause the subjects difficulty in remembering what they have seen is due to them seeing each program demonstrated one after the other, a

session which took up to an hour to complete. Apart from the factor of fatigue the subjects may have confused the run-time actions from different programs.

In summary this experiment shows that APT has improved the ability of subjects to answer the questions in the questionnaire. This suggests that APT has improved the subjects conceptual model of how the Prolog interpreter works. In addition the APT experiment has shown that further work is needed to determine how APT helps different categories of user, and which concepts form Prolog execution it clarifies and which it does not.

## 7.7 Conclusion

The interpretation of the answers given by the subjects, in the experiment to determine their misconceptions, might not be 100% correct but what is certain is the fact that these subjects have a lack of understanding of several basic concepts in Prolog which are dynamic in nature. These concepts range from how variable binding happens; the order of database search; unification of goals to the use of rules at run-time.

This evidence corroborates the findings reported in chapter 2 that many of the problems novices face in learning programming concepts are due to an incomplete or non existent conceptual model of the dynamic nature of these concepts.

The experimental study which looked at how APT affects the novices conceptual model of the action of Prolog program at run-time, demonstrates that after being given a view of program execution an improvement can be seen in the ability of novices to solve queries to Prolog programs. The results from the control group shows that this improvement is probably due to the animated view of program execution and rather than any other factor. Overall this suggests that these novices also have an improved conceptual model of the action of the Prolog interpreter due to the view of program execution provided by the APT demonstrations.

With respect to individual subjects the results show that APT may not help all novice users and may not clarify all the concepts inherent in Prolog execution. Also the results do not show what effect APT has had on the novice's conceptual model of Prolog execution. Therefore more detailed experiments are necessary in order to determine (i) how the conceptual model has changed after the demonstrations had been seen; (ii) which categories of user APT helps; (iii) which features of Prolog execution APT clarifies.

This experiment supports the hypothesis that, 'Students will be able to learn through visual examples what is actually happening in the program'. That students can learn the action of the interpreter from APT supports the hypothesis that the design principles upon which it is built present a good basis for presenting program execution to novice programmers.

# CHAPTER 8

# DISCUSSION

## 8.1 Achievements

The research described in this thesis has presented a new approach to viewing program execution, which allows novice programmers to easily assimilate a conceptual model of the action of programs at run time. This new approach to viewing program execution, called 'animated program tracing', is aimed at solving the problems novices face when learning a programming language. Both the experimental studies reported in chapter 2 and chapter 6 show that these problems are concerned mainly with the poor conceptual model novices have of the dynamic features that occur in progams when they are run. APT's (Animated Program Tracer) approach is to provide novices with a concrete base upon which to build a conceptual model of program execution.

APT is based on design principles extracted both from the general systems design principles and other tools that have made an attempt to provide a run time view of program execution. The resulting set of design principles are directed at building animated tracing tools for novice programmers. The approach that the design principles propound is to provide the novice with a clear and consistent view of program execution, showing the evaluation sequence of a program (the trace-time code) in terms of the edit-time code. The trace-time code is associated with the edit-

time code so that the novice may see and understand the relationship between the static form of the program (which is written in the editor by the user) and the dynamic form of the program (which is normally run hidden within the computer). All the dynamic features that occur in the program at run time are shown in the context of the trace-time code and also associated to the edit-time code, i.e. variable binding, backtracking, unification, recursion, s-expression evaluation, and side-effecting (the principles cope with side effects, however at this stage of implementation APT does not). This approach allows the novice programmer to follow what will happen to his/her program when it is run at the time s/he is writing it in the editor, instead of hoping for the best using a trial and error paradigm.

The three prototype APT-0 (Animated Program Tracer) systems built for the languages Prolog, Lisp and 6502 Assembler embody the above mentioned design principles, and demonstrate that these principles are applicable not only to one particular programming language but to both high/low level languages, and procedural/declarative languages. The design principles should be applicable to building animated tracing tools for all types of programming languages.

The studies reported in chapter 5 demonstrated that, in general, novice programmers understood the approach taken to viewing program execution presented by the prototype systems. Specifically, the experiment pointed out that the prototypes contained some deficiences which caused the subjects problems in understanding some parts of the trace display. This resulted in the following five lessons being learned:

1) All stepper frames must have a status line message commenting on the display.

2) The message contained in the status line must be explicit, and refer specifically to the information contained by the frame. A general message is not sufficient for novice programmers.

3) Care must be taken to ensure that terms used in the status line message have one meaning only. If they have more than one connotation then they become confusing, and loose their usefulness as a method of communication.

4) Do not impose a structure on the presentation of information to the user. For example in the Lisp prototype the structure of the 'COND' function was shown before it was evaluated. This can destroy the consistency of the display, and result in confusion.

5) The method used for showing the absence of information in the display, should not consist of highlighting the information that is present. This is not consistent with the use of highlighting. The status line should inform the user that a particular piece of information is not present.

The APT system built for Prolog allowed the above improvements to be made to the design of the prototype systems, and experiments to be run on novice programmers in order to determine whether this approach, and the design principles it is based upon, improve the understanding of the action of programs at run time by novices.

The study on novice programmers misconceptions about program execution (section 7.4) showed six main categories of misconception, and others that could not be clearly classified. The six categories are:

1) Static match

2) All subgoals must succeed

3) Real world fallacy

4) Do not search from the top

5) Only try the first rule

6) No backtracking

Of the six categories two are concerned with unification (static match, all subgoals must succeed); three are concerned with the control structure of the intepreter (do not search from the top; only try the first rule; no backtracking); and one is to do with the interference of the user's real world knowledge (reald world fallacy). The other misconceptions which could not be classified were all to do with dynamic features such as variable binding; database search; and backtracking.

These results support the notion, stated in chapter 2, that many of the misconceptions that novice programmers have concern those aspects of the

programming language that are dynamic in nature, and occur at run-time. This supports the idea of solving these misconceptions by providing the user with a view of program execution with the design features of APT.

It is clear from the experiment to determine the effect of APT on novice programmers, reported in section 7.6, that APT does improve novices ability to understand the run-time action of programs. The study shows that the group of subjects who saw APT demonstrate the execution of Prolog programs, significantly improved their scores on a questionnaire containing programs and queries which the subjects had to solve. On the other hand those subjects who only saw the solutions to the programs (rather than the execution of them) did not improve their scores significantly on the same questionnaires.

It is assumed that this improvement in performance is because the novices have built a more accurate conceptual model of program execution based on the view of program execution that APT provides.

It is worth noting that the reported improvement in subjects performance in understanding program execution found in the experiment came after only one session of demonstrations. This session, lasting about one hour, consisted of a demonstration of the execution of eight Prolog programs and queries, where the subjects saw each program being executed only once. If the subjects had been allowed to see the demonstrations more than once, or in a series of shorter sessions, their improvement would probably have been higher still.

It may be concluded that the approach to viewing program execution provided by APT and its design principles allow novice programmers to build a more accurate conceptual model of what happens to programs when they are run.

## 8.2 Problems

The APT system for Prolog embodies almost all of the design principles stated in chapter 3, with the exception of the following, 3b) side effect visibilty, 5) the integration of the interpreter's error messages into the trace-time code, and 13) a hard copy facility. The reason that these features have been left out of APT for Prolog is due soley to the lack of time, and there should be no difficulty in incorporating these features into APT. For the purposes of the experiment to determine the affect of APT on novice programmers, the lack of these features made no difference. The demonstrated programs had no errors or side effects in them, and there was no need for hard copy. These features would however be essential for novices carrying out program development, where errors and side effects crop up all too frequently.

The APT display did not show variables being renamed during program execution. The abscence of this feature can make the story of program execution confusing, especially in tracing recursive programs. Renaming of variables could easily be added to APT's display of Prolog execution. This would entail an additional step in the story where variables are renamed by adding numbered subscripts to the name of the variable each time a new goal is tried.

## 8.3 The Problems Novices have with Program Execution

Up until now it has been difficult to determine the features of programming languages and programming techniques that cause novice programmers problems when they learn how to program. The method most used today is to take verbal protocols of novices explaining, developing or debugging programs, and reading manuals. This provides useful and informative data, but it takes an inordinate amount of time to transcribe the protocols, and there are few people skilled in the art of analysis of protocols. Even then the experts seldom agree upon an interpretation.

Systems such as APT will not only help to solve some of the novices problems, but it will allow a different type of data to be gathered concering the difficulties novices have with language features, and programming techniques. Researchers will be able to study the way novices interact with APT in studying, developing and debugging programs. APT like systems should be able to automatically generate timing data for the length of time subjects spend looking at each step of a program. These timing studies could then be analysed in a similar manner to those reported in chapter 5, where stepper frames depicting similar features, i.e. variable binding, can be grouped together to find a mean viewing time for that feature. Likewise this could be done for frames depicting programming techniques.

This analysis should show up those areas of program execution which the subjects look at for a long time and those they look at for a short time. The analysis can therefore indicate which parts of program execution were found relatively hard and which were relatively easy. The conclusion drawn from this analysis should tell both which parts of program execution subjects find difficult to understand, and which parts of the stepper display are inadequate in communicating execution information. It should also be possible to use this method of evaluating programming systems to study the strategies novices use when debugging and developing programs by watching which elements of program execution they study over a period of time.

This approach means that researchers with less experience than that of those who carry out protocol analysis can run and analyse experiments concerning user misconceptions of programming languages, which will provide much needed information for the design of future languages. However a combination of protocol analysis and timing data will help ensure that a miss interpretation of either type of data does not occur.

## 8.4 Further Experiments and Research

This section suggests further work which may be carried out to enhance the knowledge already gained from the work described in this thesis. This includes , (i) experiments based on those described in previous chapters,and (ii) research concerning issues which have arisen out of experiments alreay carried out

### *8.4.1 Prototype Data Collection*

The data used to evaluate the prototype steppers (see chapter 5) consisted of both verbal protocols and timing information. The timing information was an analysis of the time each subject spent looking at each frame of the stepped display. Mean and median times were calcualated for categories of frames contained in the display and were used to determine which categories of frames subjects spent a long/short time viewing. This was taken as an indication that the subject found the informatin contained in that category of frames hard/easy to understand.

However the interpretation of the protocol data on one occasion contradicted the timing data. This highlighted a flaw in the timing analysis. Because subjects could move back and forth in the APT-0 display it was possible for any frame to be viewed more than once. Those frames which were viewed many times should have been analysed so that the viewing times were added together to produce one large time for that frame. Instead the analysis would produce a series of frames each with a relatively short viewing time. This results in low mean and median times for the category where a higher mean and median time should be found.

Future studues of this type, or any replication of the study in chapter 5, should analyse the timing data so as to take into account those frames which are viewed more than once. In summary the times from these frames should be added together before any further analysis is carried out. Also frames viewed many times should be

highlighted as this information on its own indicates that subjects found the information contained in them difficult to understand.

### *8.4.2 Explanations Experiment*

The explanations experiment (chapter 7) looked at how well subjects could explain the rules contained in Prolog programs. It was predicted that subjects would find it easier to explain programs with everyday meanings than those without, ie concrete vs abstract.

This experiment used six programs, four concrete and two abstract, and only one of the abstract and one of the concrete programs were matched for complexity. Future experiments would benefit from a much larger sample of concrete and abstract programs which are matched for static and run-time complexity. This would enable statistical tests to be applied to the resulting data to determine whether there is any difference between the number of explanations subjects got correct for each condition. This is an important area for future research as results would help the debate as to whether novice Prolog programmers should be taught using abstract or concrete programs.

### *8.4.3 Misconception Experiment*

The misconceptions experiment (chapter 7) attempted to categorise the misconceptions novice programmers hold concerning program execution. Subjects were given Prolog programs and a query and asked to solve them. Answers which were the same were grouped together and were then studied to determine the strategy of execution that could have been used to generate the answer.

It is felt that although this is a useful way to gather data, it would benefit from a related in depth study of a small number of subjects carrying out the same experiment.

The type of study recommended for this study is taking verbal protocols of subjects solving programs and queries, in the same manner as Jones (1984) and Kahney (1982). This would enable their answers to be analysed in detail to determine the exact strategies they are using to solve the problems, and thus show what model of program execution they possess. So instead of categories of misconception being imposed upon the solutions given by the subjects, evidence from the protocols could be used both to suggest categories and specify them in greater detail. These answers could then be mapped onto the answers provided by the first large scale study of novice programmers to provide a reliable analysis of a large population.

Protocol analysis would help to interpret the obscure answers, which seemed to have no sensible or obvious interpretation, and were left to reside in the miscellaneous category.

The experiment could be extended to determine whether the abstract/concrete nature of a program effects the misconceptions novices hold about program execution. As in section 8.4.2. this would require a much larger sample of concrete and abstract programs which are matched for static and run-time complexity. The result from such an experiment would also help the debate on the use of concrete and abstract programs as a medium for teaching novices Prolog.

If the experiments suggested in this section and section 8.4.2 were carried out it would allow a comparison to be made of the ability of novices to explain and solve Prolog programs. This correlation was attempted in section 7.5 but due to the small sample of programs used a statistical test was deemed to be of little use. However an increased sample size would allow such a statistical correlation test to be applied producing a more reliable interpretation of the results than an eyeball analysis.

### *8.4.4 APT Experiment*

The experiment described in section 7.6 attempted to determine the effect of APT demonstrations on the ability of novice programmers to solve Prolog programs. The method used in this experiment was fairly tight although certain areas were slack and could be made more rigorous. The following areas of the experiment could be improved.

The number of subjects used was small, only six in each condition. A larger sample would produce more reliable results. The subjects used for each condition although matched for programming experience were not from the same population pool. Future experiments should use subjects from the same pool for both conditions.

The subjects carried out no pretest to determine their ability to solve problems and program. If a pretest had been carried out, and a large enough sample of subjects used, the subjects could be split into groups of differeing ability. This would allow, (i) subjects of differing ability to be spread evenly across both conditions and (ii) a test of how useful APT is to novice programmers of differing programming ability.

Likewise, a larger number of programs containing different aspects of Prolog would allow a test of how useful APT is in communicating run-time information for different aspects of Prolog.

The length of time subjects spent on the experimental and control conditions was not martched. This could have allowed fatigue and practice effects to creep into the results unnoticed. Future experiments should match the length of time spent on different conditions to prevent such errors and ensure that experimental effects are due to the dependent variable rather than side effects of the experiment.

### *8.4.5 Other Experiments*

Apart from the hypothesis that 'students will be able to learn through visual examples what is actually happening in the program', which was tested in chapter 7, there were two other predictions made concerning the effect of the approach of APT on novice programmers.

1) Students will be able easily to debug their mistakes by understanding them in terms of the evaluation of the code.

2) Students will be able to develop their programs more quickly due to the ease of monitoring the surface evaluation of the program.

The above section describes ways in which the facilities that APT provides can be used to test the above predictions.

Experiments to test these predictions were not carried out here because they do not fall within the scope of this thesis, which is to develop a more systematic basis for the design of animated tracing tools. The experiments reported in chapter 4 and 7 were carried out to in order to determine the success of the design principles, embodied in both APT-0 and APT, in communicating run-time information to novice programmers.

## 8.5 Integration of APT into a Tutoring Environment

A dynamic view of program execution that is defined by the design principles stated in chapter 3, and provided by APT provides a new approach for the basis of explanation in automated tutoring environments. There are many tutoring systems and program analysers which have the ability to trap, analyse and understand errors made by novice programmers (Ruth, 1976; Waters, 1979; Eisenstadt and Laubsch, 1980; Domingue, 1985; Johnson and Soloway, 1985; Murray, 1986). However once they have found and analysed an error such systems tend to provide little explanation other than canned text, from which novices have difficulty hypothesising

what their error was and how to correct both it and their faulty conceptual model of program execution.

If a system such as APT were embedded into a tutoring environment it would allow the tutoring system to provide a graphic illustration of programming language features, or programming techniques, both to explain the correct model of program execution and to demonstrate to the novice how and why his/her program is faulty. This approach to explanation should allow the novice to speedily understand the error made, to correct his/her conceptual model of program execution, and to introduce him/her to the debugging facilities provided for novice programmers.

Animated views of programming environments need not be confined to languages. The design principles could equally well be applied to other dynamic systems which are complex and difficult to learn, for example operating systems, programmable robot control, and just about any interactive computer environment.

## 8.6 A Debugging Environment for Novices and Experts

A large problem for computer systems designers is designing an environment that will both be easy to use and learn for novice users, and efficient for expert users. In the past this has meant that one of the groups has been ignored, so that the system is either very easy to use, takes a long time to do anything and is not particularly powerful , i.e. SOLO (Eisenstadt, 1983), or it is complex, difficult to learn but powerful and efficient in the hands of experts, i.e. UNIX™. This ultimately means that as users progress from novices to experts, or from less adept users to more adept, they are forced to leave behind systems or tools they have used previously and learn new more powerful and efficient tools.

In the domain of program debugging, or viewing program execution, the approach propounded by APT should help bridge the divide between novices and experts. In its present state APT presents a very detailed view of program execution

which is ideal for novices, but not for experts. It would however be simple to alter the view presented so that it would suit experts. The grain of detail shown to the user could be made user definable, and interactive, allowing the user to zoom in on the error made and then view the error in the desired detail. Also because such systems as APT need to be knowledge based in order to provide the amount of detail for the novice it should be possible to design an interactive query system allowing the user to ask question of the debugger so that hypotheses concerning the error can be rapidly tested.

It should also be possible to link APT like systems to other more specialized views of program execution such as graphs and stacks, and may even provide a method of introducing these views to new users.

## 8.7 Summary

This thesis is concerned with the principled design of a computational environment which depicts an animated view of program execution for novice programmers. The approach taken is that being formed by the new discipline of 'Human-Computer Interaction', which is exemplified by the journal of the same name. This interdisciplinary approach has as yet not developed its own methodology but borrows freely from the fields of 'Artificial Intelligence', 'Cognitive Psychlogy', and 'Computer Science' in an attempt to improve the state of knowledge about computer interfaces based on the perceived problems encountered by computer users. The aim of this thesis then, is to develop a more systematic, if not yet scientific, basis for the design of animated tracing tools.

We began by stressing the importance of tracing tools in showing the detail of what happens during program execution, and asserted that a principled animated view of program execution should benefit novice programmers by: (i) helping students conceptualize what is happening when programs are executed; (ii) simplifying

debugging through the presentation of bugs in a manner which the novice will understand; (iii) reducing program development time.

A survey of the literature from three different fields showed: (i) the problems that novices encounter when learning a programming language; (ii) the general design principles for computer systems; (iii) a critical appraisal was given of various systems which attempt to solve the above problems by presenting a view of program execution.

From this review a set of design principles was extracted for the design of an animated view of program execution. These principles were given the following names:

*Edit-time and trace-time code isomorphism* - the code used to show the trace is based upon the code the user has typed into the editor.

*In-place subroutine instantiation* - called/matched pieces of code are inserted into the trace-time display as they are called/matched.

*What You See Is What Happens* - 'WYSIWHa' - this is the notion of showing the virtual machine in action, providing novices with a concrete view of program execution. This consists of showing, the evaluation of code as it happens; the occurance of side-effects; the binding of variables in the context of the trace-time code.

*Description level of trace* - The description level of the trace should reflect the goal of the user. For novice programmers the description level of the trace should be the outward appearance of the language stated in the language specification, and independent of implementation specific details.

*Status line navigation* - brief messages, contained in a status line, comment on the state of the trace at each step provide the novice with enough information to either remind him/her what is happening or direct him/her to a more informed source.

*Trace forwards and backwards* - the view of program execution should allow the user to run it both forwards and backwards so that a tricky piece of code can be reviewed without the tracer being reinvoked.

*Integration of the interpreter's error messages* - error messages should be integrated into the trace-time code to provide information concerning the events leading up to the error.

*Uniformity of the editor, top-level and utilities* - the tracing system and the top-level should be integrated with the editor to reduce the amount of information necessary for the novice to be able to use the environamnt.

*Demonstration utility* - at their most detailed level the tracer should be able to be used to demonstrate different features of the programming language which are particularly difficult for novices to learn.

*Minimal Extraneous symbols* - as few symbols as possible should be used to flag features in the tracing environment. The presence of these symbols not only clutters up the screen, but increases the amount the novice needs to know before s/he can use the system. The information provided by the status line in conjunction with inverse video highlighting or colour should flag these features more than adequately.

*Non proliferation of views* - The number of views of program execution should be kept to a minimum for novice programmers.

*Detail/speed trade off* - Novice programmers need a slow inflexible detailed story of program execution which will enable them to concentrate on building a conceptual model rather than on how the tracing environment works.

*Display shape* - The shape and placement of windows displaying the trace should be determined by the natural shape of the code for each programming language.

These design principles (with the exception of 'integration of the interpreter's error messages') were embodied in three 'canned' stepper displays for Prolog, Lisp and 6502 Assembler. These prototypes, called APT-0 (Animated Program Tracer), demonstrated that the design principles can be broadly applied to procedural and declarative; low and high level languages. An experimental study looking at how long subjects took to view each step of the programs presented in the canned displays suggested the following improvements to APT-0.

1) All stepper frames must have a status line message commenting on the display.

2) The message contained in the status line must be explicit.

3) Care must be taken to ensure that terms used in the status line message have one meaning only.

4) Do not impose a structure on the presentation of information to the user.

5) The method used for showing the absence of information in the program, should not consist of highlighting the information that is present.

These improvements were incorporated in a real implementation of APT for Prolog. Prolog was chosen over the other languages as it has relatively unsophisticated tracing tools available for its users, and has several very tricky concepts which novices find difficult to understand, i.e. backtracking and unification. APT, uses an object centred representation, is built on top of a Prolog interpreter and environment, and is implemented in Common Lisp and Zeta Lisp and runs on the Symbolics™ 3600 range of machines. This principled approach embodied by APT provides two important facilities which have previously not been available, firstly a means of demonstrating dynamic programming concepts such as variable binding, recursion, and backtracking, and secondly a debugging tool which allows novices to step through their own code watching the virtual machine in action. This moves towards simplifying the novice's debugging environment by supplying program execution information in a form that the novice can easily assimilate.

An investigation into the misconceptions novices hold concerning the execution of Prolog programs showed that the order of database search, and the concepts of variable binding, unification and backtracking are poorly understood. A further study was conducted which looked at the effect that APT had on the ability of novice Prolog programmers to understand the execution of Prolog programs. This demonstrated that the performance of subjects significantly increased after being shown demonstrations of the execution of Prolog programs on APT, while the control group who saw no demonstration showed no improvement.

The experimental evidence demonstrates the potential of APT, and the principled approach which it embodies, to communicate run-time information to novice programmers, increasing their understanding of the dynamic aspects of the Prolog interpreter.

## REFERENCES


Anderson, J, R., Farrel, R., and Sauers, R. Learning to plan Lisp. Pittsburgh, PA:, Department of Psychology, Carnegie-Mellon, 1982.

Anderson, J, R., Farrel, R., and Sauers, R. Learning to Program in Lisp. Pittsburgh, PA:, Department of Psychology, Carnegie-Mellon, 1983.

Anderson, J, R., Pirolli, P., and Farrell, R. Learning to Program Recursive Functions. In M. Chi, R .Glaser, and M. Farr (Eds) The Nature of Expertise. Hillsdale, New Jersey: Erlbaum. 1984.

Baeker, R. Two systems which produce animated representations of the execution of computer programs. SIGCSE Bulletin, 7, 158-167, 1975.

Barr, A., Beard, M., and Atkinson, R.C. The computer as a tutorial laboratory: The Stanford BIP Project. IJMMS, 8, 567-596, 1976.

Brooks, R. Towards a theory of the cognitive processes in computer programming. IJMMS, 9, 1977.

Bundy, A. What stories should we tell Prolog students? D.A.I. Research Note 198. Department of Artificial Intelligence, University of Edinburgh, 1983.

Bundy, A., Pain, H., Brna, P., and Lynch, L. A Proposed Prolog Story. D.A.I. Research Paper 283. Department of Artificial Intelligence, University of Edinburgh, 1986.

Byrd, L. Understanding the Control Flow of Prolog Programs. Proceedings of the Logic Programming Workshop, Ed S. Tarnlund. p 127-38, 1980.

Burke, S.G., Carrette, G.J, and Eliot, C.R. NIL Reference Manual, Massachusetts of Technology, MIT LCS Publication, 1984.

Card, S.K., Moran, T.P., and Newell, A. The Psychology of Human-Computer Interaction, Lawrence Erlbaum, New Jersey, 1983.

Carroll, J, M. and Thomas, J, C. Metaphor and the cognitive representation of computing systems. Report RC 8302, IBM Watson Research Center, May. 1980.

Collins, A.M., and Quillian, M.R. Retrieval Time from Semantic Memory. Journal of Verbal Learning and Verbal Behaviour. Vol 8, pp240-7, 1969.

Coombs, M.J., Hartley, R.T., and Stell, J.G. Debugging User Conceptions of Interpretation Processes. Proceedings of AAAI-86, Fifth National Conference on Artificial Intelligence, August 11-15 Philadelphia, PA, 1986.

Clocksin, W.F., and Mellish, C.S. Programming in Prolog, Springer-Verlag, 1981.

Delisle, M.N., Menicosy, D.E, and Schwartz, M.D. Viewing a Programming Environment as a Single Tool. Proceedings of the ACM

SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. vol 4, no 3, 1984.

Dionne, M.S., and Mackworth, A.K. ANTICS: A system for animating Lisp programs. Computer Graphics and Image Processing, 7, 1, 1978.

di Sessa A. A. A Principled Design for an Integrated Computational Environment. MIT Technical Report, 1982.

di Sessa A. A. A Principled Design for an Integrated Computational Environment. Human-Computer Interaction, vol 1, pp1-47, 1985.

Domingue, J. Towards an Automated Programming Advisor. Technical Report No. 16, Human Cognition Research Laboratory, The Open University, Milton Keynes, England, 1985.

du Boulay, J.B.H., O'Shea, T., and Monk, J. The Black Box Inside The Glass Box: Presenting Computing Concepts To Novices. IJMMS, 14, 3, 237-249, 1981.

Eisenstadt, M., and Laubsch, J. Towards an Automated Debugging Assistant for Novice Programmers. Proceedings of the AISB-80 Conference on Artificial Intelligence, July 1980.

Eisenstadt, M. A User Friendly Software Environment for the Novice Programmer. Communications of the ACM, vol 26, no 12, Dec 1983.

Eisenstadt, M. A powerful Prolog trace package. Proceedings of the Sixth European Conference on Artificial Intelligence, North-Holland, 1984.

Eisenstadt, M., Breuker, J., Evertsz, R. Mundane Expertise In Iteration. Technical Report No.7, Human Cognition Research Laboratory, The Open University, Milton Keynes, England, 1984.

Eisenstadt, M., Hasemer, T., and Kriwaczek, F. An improved user interface for Prolog. Proceedings of the IFIP Conference on Human Computer Interaction. North-Holland, 1984.

Eisenstadt, M. D309 Artificial Intelligence Project. Unit of Cognitive Psychology: A third level course. Milton Keynes: Open University Press, 1986.

Gugerty, L., and Olson, G.M. Comprehension Differences in Debugging by Skilled and Novice Programmers. In E. Soloway and S. Iyengar (Eds.) Empirical Studies of Programmers, Papers presented at the First Workshop on Empirical Studies of Programmers June 5-6. Norwood, New Jersey, Ablex. 1986.

Halasz, F., and Moran, T. Analogy Considered Harmful. Proceedings of Human Factors in Computer Systems Conference, National Bureau of Standards, Gaithersburg, Maryland, 1982.

Hasemer, T. An Empirically-Based Debugging System for Novice Programmers. Technical Report No.6, Human Cognition Research Laboratory, The Open University, Milton Keynes, England, 1983.

Hasemer, T. A Beginner's Guide To Lisp. Addison-Wesley, 1984.

Johnson, L.W., and Soloway, E. PROUST: An Automatic Debugger for Pascal Programs. BYTE, April, vol 10, No. 4, pp179-180, 1985.

Jones, A. How Novices Learn to Program. Proceedings of INTERACT '84. 1984.

Jones, A. Learning to Program: Some Protocol Data. Technical Report No.41, Computer Assisted Learning Research Group, The Open University, Milton Keynes England, 1984

Kahney, J.H. An in-depth study of the behaviour of novice programmers. Technical Report No.5, Human Cognition Research Laboratory, The Open University, Milton Keynes, England, 1982.

Lewis, M.W. Improving SOLO's User-interface: an Empirical Study of User Behaviour and a Proposal for Cost Effective Enhancement.. Technical Report No. 7, Computer Assisted Learning Research Group, The Open University, Milton Keynes, England, 1980.

Lieberman, H. Steps Toward Better Debugging Tools For Lisp. Proceedings ACM Symposium on Functional Programming, 1984.

Lukey, F. J. Understanding and Debugging Programs. IJMMS, 12, 189-202, 1980.

Mayer, R. E. Some conditions of meaningful learning for computer programming: advance organizers and subject control of frame order. Journal of Educational Psychology, 68, (2), 143-150. 1976.

Mayer, R.E. A psychology of learning BASIC. Communications of the ACM, 22, 589-593, 1979.

Mayer, R. E. The Psychology of How Novices Learn Computer Programming. Computing Surveys, vol 13, no 1, 1981.

Miller, L.A. Programming by Non-programmers. International Journal of Man Machine Studies 6, 237-260. 1974.

Murray, W.R. Automatic Program Debugging for Intelligent Tutoring Systems. Technical Report AI TR86-27, The University of Texas at Austin, Texas, USA, 1986.

Norman, D A. Some observations on Mental Models. In D. Genter and A. Stevens (Eds.) Mental Models. Hillsdale, New Jersey, Erlbaum. 1982.

Norman, D.A. Design Principles for Human-Computer Interfaces. Proceedings of the CHI 1983 Conference on Human Factors in Computer Systems. Boston: December 1983a.

Norman, D.A. Design Rules Based on Analyses of Human Error. Communications of the ACM, 26, 254-258, 1983b.

Pain, H., and Bundy, A. What Stories Should we Tell Novice Prolog Programmers? In Hawley (ed) Artificial Intelligence Programming Environments Book, 1985.

Paxton, A.L., and Turner, E.J. The application of human factors to the needs of the novice computer user. IJMMS, 20, 137-156, 1984.

Pirolli, P.L., and Anderson, J.R. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. Canadian Journal of Psychology. 1985.

Plummer, D. SODA Screen Oriented Debugging Aid. D.A.I Research Note 260, Department of Artificial Intelligence, University of Edinburgh, 1985.

Rajan, T. APT: The Design of Animated Tracing Tools for Novice Programmers. Technical Report No. 15, Human Cognition Research Laboratory, The Open University, Milton Keynes, England, 1985.

Reiss, S.P. Graphical Program Development with PECAN Program Development Systems. ACM SIGPLAN Notices 19, 5, 1984a.

Reiss, S.P. PECAN: Program Development Systems that Support Multiple Views. Proceedings of the 7th International Conference of Software Engineering. 1984b.

Roberts, T.L. and Moran, T.P. The Evaluation of Text Editors: Methodology and Empirical Results. Communications of the ACM, Vol 26, No. 4, April 1983.

Rumelhart, D, E. and Norman, D, A. Analogical processes in Learning. In J. R. Anderson (ed), Cognitive Skills and Their Aquisition, Hillsdale, New Jersey: Erlbaum. 1981.

Ruth, G.R. Intelligent Program Analysis. Artificial Intelligence, 7, pp 65-85, 1976.

Sheil, B.A. The Psychological Study of Programming. Computing Surveys of the ACM, vol 13, No. 1. March 1981.

Sime, M.E., Green, T.R.G., and Guest, D.J. Psychological evaluation of two conditional constructions in computer languages. IJMMS, 5, 123-143, 1973.

Sime, M.E., Arblaster, A.T. and Green, T.R.G. Reducing Programming Errors in Nested Conditionals by Prescribing a Writing Procedure. International Jounal of Man Machine Studies 9,119-126. 1979.

Smith, D.C., Irby, C., Kimball, R., and Verplank, B. Designing the Star User interface. BYTE, April vol 7, pp242-282, 1982.

Soloway, E., Boner, J., and Ehrlich, K. Cognitive factors in programming: an empirical study of looping constructs. Amherst, MA: Technical Report No.81-10, Department of Computer Science, University of Massachussetts, 1981.

Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. What do Novices Know about Programming. In B. Schneiderman and A. Badre (ed), Directions in Human-Computer Interactions. Ablex. 1982.

Soloway, E., and Ehrlich, K. Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering. Special Issue: Reusability, Sept. 1984.

Teitelbaum, T., and Reps, T. The Cornell Program Synthesizer: A Syntax Directed Programming Environment. Communications of the ACM, vol 24, no 9, 1981.

Touretzky, D.S. Lisp, A Gentle Introduction to Symbolic Computation. Harper and Row, 1984.

van Someren, M, W. Miconceptions of Beginning Prolog Programmers. Memorandum 30 of the Research Project, 'The Acquisition of Expertise'. Dept. of Experimental Psychology. The University of Amsterdam. 1984.

van Someren, M, W. Beginners Problems in Learning Prolog. Memorandum 31 of the Research Project, 'The Acquisition of Expertise'. Dept. of Experimental Psychology. The University of Amsterdam. 1985.

Waters, R.C. A method for analyzing loop programs. IEEE Transactions on Software Engineering, SE-5:3, May, 1979.

Weinreb, D., and Moon, D. Lisp Machine Manual. Symbolics Inc. 1981.

Winston, P.H., and Horn, B.K.P. LISP, Adison-Wesley, 1981.

Winston, P.H., and Horn, B.K.P. LISP, Second Edition. Adison-Wesley, 1984.

Young, R, M. Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive Devices. In J. R. Anderson (ed), Cognitive Skills and Their Aquisition, Hillsdale, New Jersey: Erlbaum. 1981.

# APPENDIX-A

Appendix A presents questionnaire 1, comprising eight Prolog programs, a one page introduction, and two examples of how the subjects should answer the questions. This questionnaire was used in several of the experiments reported in the thesis. In the explanations study the questionnaire was used without the fourth program. In the misconceptions study the questionnaire was used in its entirety. This questionnaire was also used in the first test in the experiment to determine the affect of APT on novice programmers.

The correct answers to the questionnaire are shown below each query in this font.

Dear D309 student,
As a full-time Ph.d. student in Psychology at the Open University, my only access to large numbers of subjects is through questionnaires like this one. My research concerns problems faced by novice Prolog programmers, hence my interest in D309 students. Hopefully my findings will enable us to determine what improvements can be made to D309 for the benefit of future students. In order for this experiment to be successful you must not ask your colleagues for help, nor copy their answers.

One or two of you will have already done this experiment. If this is the case then please disregard this letter, and I'm sorry to have bothered you.

Enclosed you will find several sheets of paper. Each sheet of paper has written on it a Prolog program, and a Prolog query (a question that you ask Prolog to answer). Some of the Prolog programs are simple; others more difficult. Carefully read each program and answer the query in the same way that Prolog would if the program and query had been typed into the computer. Some of the programs ask you to give all the solutions to the Prolog query, this is clearly marked with **'A's:'** prompt instead of the usual **'A:'** prompt.

Inside there are two sheets of paper providing you with a model answer (one at the front and one towards the end), showing how you should go about answering these questions.

**Note: This experiment will not affect your course grades.**

Below you will find a couple of questions, and a short tutorial on the PRINT statement 'PP'. Measure how much time you spend on the experiment, and write it in the box below, but do not spend more than an hour in total.

**It is most important for the purposes of this experiment that you return these sheets to me as soon as possible, for example within a couple of days.**

Thank you for your help.
Tim Rajan

Did you do the A.I. project at Summer School YES NO

Do you have any experience of other programming languages

What is the total time you spent on this experiment

Your name (OPTIONAL)

# TUTORIAL ON 'PP' - PLEASE READ FIRST

## This is not part of the experiment

This is a short tutorial on the print command in Prolog i.e. **'PP'**. This command is used to print information to the screen when a program is running. **PP** can take any number of arguments (the items in between the brackets), both variables and constants, for example here are some print statements in plain text and the values they would print out in bold (for this example assume that the variable _X has the value **table**)

| | |
|---|---|
| PP(john). | **john** |
| PP(_X). | **table** |
| PP(john is sitting at the _X). | **john is sitting at the table** |

Print commands can take any number of variables as arguments as well, i.e. PP(_X _Y _Z).

Please keep an eye out for the PP's in the programs, and write down the values they print as you go along. You could use a different coloured pen to differentiate the variable bindings that Prolog prints out at the end of a program from the values that are printed from a PP command.

***AN EXAMPLE OF HOW YOU SHOULD ANSWER THESE PROBLEMS***

*All answers are shown in italics*
ALL COMMENTS ARE SHOWN IN CAPITALS

```
likes(Tony _Y) if
   drinks(_Y wine) &
   PP(_Y) &
   tall(_Y).

drinks(Jon wine).
drinks(Hank wine).

tall(Hank).
```

**Q: likes(Tony _Y).**

**A's:**

| | |
|---|---|
| *Jon* | PRINTED OUT BY THE LINE PP(_Y) |
| *Hank* | PRINTED OUT BY THE LINE PP(_Y) |
| *_Y = Hank* | PRINTED OUT AS A SOLUTION |

*no (more) solutions*

```
likes(mary  wine).
likes(mary  food).
likes(john  food).
likes(john  mary).

bothlike(_X)  if
   likes(mary  _X)  &
   PP(_X)  &
   likes(john  _X).


Q:  bothlike(_X).

A: wine
   food
   _X = food
yes
```

```
has(fred  money).
has(joe  money).
has(james  money).

kisses(jane  fred).
kisses(june  james).

likes(_X  _Y)  if
   has(_Y  money)  &
   PP(_Y)  &
   kisses(_X  _Y).
```

**Q: likes(_X _Y).**

***Note ALL answers are required***

**A's:** fred
    _X = jane _Y = fred
    joe
    james
    _X = june _Y = james
no

```
origin(BA137  Chicago).
origin(TWA194  Dallas).
origin(PA100  London).
origin(AZ129  London).

destination(TWA194  Paris).
destination(PA100  Rome).
destination(AZ129  Pisa).

stopover(BA137  Washington).
stopover(TWA194  Boston).
stopover(AZ129  Rome).

connected(_F1  _F2)  if
   destination(_F1  _X)  &
   PP(_F1)  &
   origin(_F2  _X).
connected(_F1  _F2)  if
   destination(_F1  _X)  &
   PP(_X)  &
   stopover(_F2  _X).


Q:  connected(_F1  _F2).

A: TWA194
   PA100
   AZ129
   Paris
   Rome
   _F1 = PA100 _2 = AZ129
yes
```

```
kisses(mary  john).
kisses(john  june).

hasflu(_X)  if
   PP(_X)  &
   kisses(_Y  _X)  &
   hasflu(_Y).
hasflu(mary).
```

```
Q:  hasflu(june).

A: june
   john
   mary
yes
```

```
sisters(_X _Y) if
   female(_X) &
   parents(_X _M _F) &
   parents(_Y _M _F).

female(alice).
female(june).
female(mary).

parents(sue victoria fred).
parents(alice victoria albert).
parents(mary susan edward).
parents(june victoria albert).
parents(jenny jill roy).
```

**Q: sisters(_X _Y).**

***Note ALL answers are required***

**A's:** _X = alice _Y = alice
_X = alice _Y = june
_X = june _Y = alice
_X = june _Y = june
_X = mary _Y = mary
no

DO NOT GO BACK TO THE PREVIOUS QUESTION

```
sisters(_X _Y) if
   female(_X) &
   parents(_X _M _F) &
   parents(_Y _M _F).

female(alice).
female(june).
female(mary).

parents(sue victoria fred).
parents(alice victoria albert).
parents(mary susan edward).
parents(june victoria albert).
parents(jenny jill roy).
```

**Q: sisters(alice alice).**

**A:yes**

```
a(_X) if
  b(_X) &
  c(_X).

b(_X) if
  h(_X _Y) &
  PP(_X _Y) &
  i(_Y).
b(_X) if
  h(_Y _X) &
  PP(_Y _X) &
  i(_Y).

h(john mary).
h(jim sue).

i(mary).
i(jim).

c(mary).
c(sue).
c(fred).
c(jane).
```

```
Q: a(_X).

A: john mary
   jim sue
   john mary
   jim sue
  _X = sue
yes
```

## *AN EXAMPLE OF HOW YOU SHOULD ANSWER THE NEXT PROBLEM*

*All answers are shown in italics*
ALL COMMENTS ARE SHOWN IN CAPITALS

NOTE IT IS O.K. FOR PREDICATES SUCH AS'a' AND 'b' TO HAVE NO ARGUMENTS

**a if**
**b &**
**c.**

**b if PP(Hi).**

**c if PP(Bye).**

**Q: a.**

**A:** ***Hi*** PRINTED OUT BY THE LINE PP(HI)
***Bye*** PRINTED OUT BY THE LINE PP(BYE)

***yes***

```
a if
  b &
  loves(John Mary).
a if c.

b if
  d &
  e.
b if f.

c if PP(foo).

d if PP(bar).

e if PP(baz).

e if PP(gort).

f if PP(fez).

loves(John Sue).



Q: a.


A: bar
   baz
   gort
   fez
   foo
yes
```

## APPENDIX-B

Appendix B presents questionnaire 2, comprising eight Prolog programs, a one page introduction, and two examples of how the subjects should answer the questions. This questionnaire was used as the second test in the second experiment concerning the affect of APT on novice programmers. This questionnaire is an isomorph of first questionnaire.

The correct answers to the questionnaire are shown below each query in this font.

# TUTORIAL ON 'write' - PLEASE READ FIRST

## This is not part of the experiment

This is a short tutorial on the print command in Prolog i.e. **'write'**. This command is used to print information to the screen when a program is running. **write** can take any number of arguments (the items in between the brackets), both variables and constants, for example here are some print statements in plain text and the values they would print out in bold (for this example assume that the variable _X has the value **table**)

| | |
|---|---|
| write(john). | **john** |
| write(_X). | **table** |
| write(john is sitting at the _X). | **john is sitting at the table** |

Print commands can take any number of variables as arguments as well, i.e. write(_X _Y _Z).

Please keep an eye out for the write's in the programs, and write down the values they print as you go along. You could use a different coloured pen to differentiate the variable bindings that Prolog prints out at the end of a program from the values that are printed from a write command.

***AN EXAMPLE OF HOW YOU SHOULD ANSWER THESE PROBLEMS***

*All answers are shown in italics*

ALL COMMENTS ARE SHOWN IN CAPITALS

```
likes(Tony _Y) if
   drinks(_Y wine) &
   write(_Y) &
   tall(_Y).

drinks(Jon wine).
drinks(Hank wine).

tall(Hank).
```

**Q: likes(Tony _Y).**

**A's:**

| | |
|---|---|
| *Jon* | PRINTED OUT BY THE LINE WRITE(_Y) |
| *Hank* | PRINTED OUT BY THE LINE WRITE(_Y) |
| *_Y = Hank* | PRINTED OUT AS A SOLUTION |

*no (more) solutions*

```
colour(cat  blue).
colour(cat  black).
colour(coal  black).
colour(sea  green).

samecolour(_X)  if
   colour(cat  _X)  &
   write(_X)  &
   colour(coal  _X).


Q:  samecolour(_X).

A: blue
   black
   _X = black
yes
```

```
is(harold  tall).
is(mark  tall).
is(john  tall).

livesnear(sue  harold).
livesnear(carol  john).

loves(_X  _Y)  if
   is(_Y  tall)  &
   write(_Y)  &
   livesnear(_X  _Y).
```

**Q: loves(_X _Y).**

***Note ALL answers are required***

**A's:** harold
_X = sue _Y harold
mark
john
_X = carol _Y = john
no

```
playswith(Jim   Phil).
playswith(Sue   Mark).
playswith(Fred   Jon).
playswith(Doreen   Jon).

likes(Sue   Mike).
likes(Fred   Anne).
likes(Doreen   Simon).

talksto(Jim   Pat).
talksto(Sue   Hank).
talksto(Doreen   Anne).

friends(_P1   _P2)   if
   likes(_P1   _X)   &
   write(_P1)   &
   playswith(_P2   _X).
friends(_P1   _P2)   if
   likes(_P1   _X)   &
   write(_X)   &
   talksto(_P2   _X).
```

```
Q:  friends(_P1   _P2).

A: Sue
   Fred
   Doreen
   Mike
   Anne
   _P1 = Doreen _P2 = Anne
yes
```

```
talksto(Mike  Kevin).
talksto(Kevin  Nicky).

understandsprolog(_X)  if
   write(_X)  &
   talksto(_Y  _X)  &
   understandsprolog(_Y).

understandsprolog(Mike).
```

**Q: understandsprolog(Nicky).**

**A:** Nicky
Kevin
Mike
yes

```
colleagues(_X _Y) if
   person(_X) &
   worksfor(_X _B) &
   worksfor(_Y _B).

person(Anne).
person(John).
person(Simon).

worksfor(Tim Marc).
worksfor(Anne Hank).
worksfor(Simon Bill).
worksfor(John Hank).
worksfor(Dot Jill).
```

**Q: colleagues(_X _Y).**

***Note ALL answers are required***

**A's:** _X = Anne _Y = Anne
_X = Anne _Y = John
_X = John _Y = Anne
_X = John _Y = John
_X = Simon _Y = Simon
no

DO NOT GO BACK TO THE PREVIOUS QUESTION

```
colleagues(_X _Y) if
   person(_X) &
   worksfor(_X _B) &
   worksfor(_Y _B).

person(Anne).
person(John).
person(Simon).

worksfor(Tim Marc).
worksfor(Anne Hank).
worksfor(Simon Bill).
worksfor(John Hank).
worksfor(Dot Jill).
```

**Q: colleagues(Anne Anne).**

**A:** yes

```
eligible(_X)  if
  sociable(_X)  &
  single(_X).

sociable(_X)  if
  friends(_X  _Y)  &
  write(_X  _Y)  &
  likeable(_Y).
sociable(_X)  if
  friends(_Y  _X)  &
  write(_Y  _X)  &
  likeable(_Y).

friends(John  Mary).
friends(Jim  Sue).

likeable(Mary).
likeable(Jim).

single(Mary).
single(Sue).
single(Fred).
single(Jane).
```

```
Q:  eligible(_X).

A: John Mary
   Jim  Sue
   John Mary
   Jim  Sue
   _X = Sue
yes
```

## ***AN EXAMPLE OF HOW YOU SHOULD ANSWER THE NEXT PROBLEM***

*All answers are shown in italics*
ALL COMMENTS ARE SHOWN IN CAPITALS

NOTE IT IS O.K. FOR PREDICATES SUCH AS'a' AND 'b' TO HAVE NO ARGUMENTS

```
a if
  b &
  c.

b if write(Hi).

c if write(Bye).
```

**Q: a.**

**A:** ***Hi*** PRINTED OUT BY THE LINE WRITE(HI)
***Bye*** PRINTED OUT BY THE LINE WRITE(BYE)

***yes***

```
p if
  q &
  goes(Jim shop).
p if r.

q if
  s &
  t.
q if u.

r if write(foo).

s if write(bar).

t if write(baz).

t if write(gort).

u if write(fez).

goes(Sue shop).
```

```
Q: p.

A: bar
   baz
   gort
   fez
   foo
yes
```

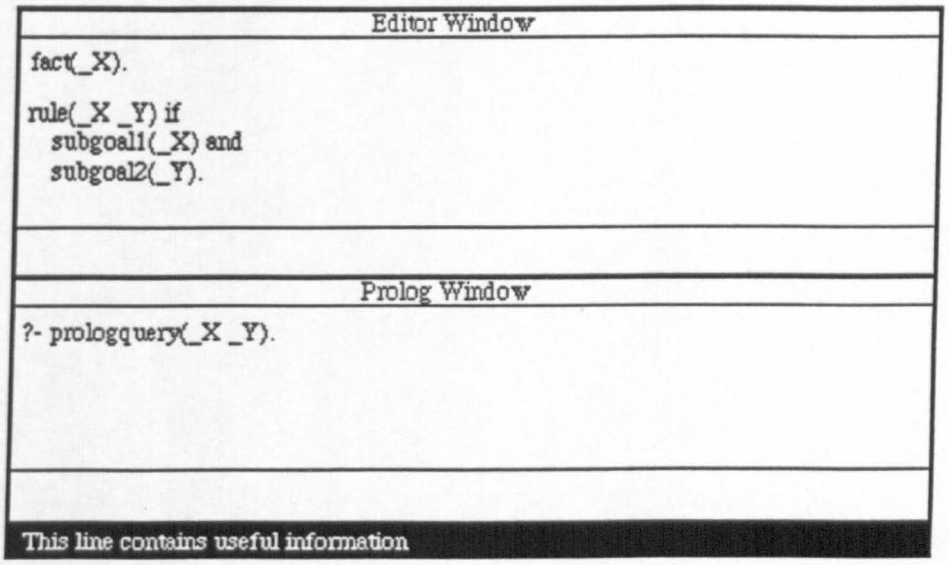
## APPENDIX-C

Appendix C presents the introduction to APT, given to the subjects carrying out the experiment to determine the affect of APT on novice programmers.

## APT

This is a short introduction to **APT**, the Animated Program Tracer. It describes the layout of the screen; what you will see when **APT** steps through a Prolog program; and what you should look out for when using the system.

The picture below shows what the APT screen will look like when you first see it.




The screen is divided into three parts, or windows. The top window called the **Editor Window** should be familiar to you. This window contains the Prolog program that you type into the computer. When editing the program you have access to all the editing facilities that you would have in a word processor. The window below the Editor window is the **Prolog Window**. This is where you ask questions (queries) of Prolog and where Prolog prints out its answers, for example variable

bindings and information printed by print commands, the print command in this Prolog is called **write** and will look something like **write(_X _Y)**, see the attached tutorial sheet. The bottom window is called the Status line.

In normal use you would write your program in the editor window, and when you had finished ask a quesion of Prolog in the Prolog window, typing the question next to the prompt that Prolog provides i.e. **"?-"**. However when **APT** steps a program the way the three windows mentioned above are used changes somewhat. What happens in each of these windows when a program is stepped is described below.

## General Description of APT

The command **step** is typed into the Prolog Window, this asks Prolog to step through the program which is run when the next Prolog query is asked. When the Prolog query is typed into the computer, APT shows the order that Prolog looks through its database (which is contained in the Editor Window) to find Prolog rules or facts that match the query. If a matching rule can be found in the database, **APT** shows how the variables get bound (matched) to values, and consequently how the subgoals of that rule (the parts of the rule that come after the **'if'**) are matched against items contained in the database. The animation of the program running is shown in the **Prolog Window**. This animation is associated with the database in the **Editor Window** using highlighting i.e. inverse video. Comments on what is happening at any particular stage of the program can be found on the **Status Line** (see bottom of picture).

## Prolog Window

After the Prolog query has been typed the Prolog Window shows the outstanding goals that have to be matched against the database. These goals or subgoals are the part of the rule following the **'if'** (see example in Prolog syntax below). These subgoals can match against the predicate or head (the part of the rule before the **'if'** or alternatively a fact) of other rules or facts in the database. When this happens the fact or rule is inserted into the display to show the outstanding goals to be matched. The display then moves down this goal list in the same manner as Prolog does. As the program is animated the outstanding goal list (display) grows accordingly to accomodate subgoals. However if subgoals fail they are removed from the goal list, and the display may shrink. All the important points to be seen are shown using highlighting, and are commented on via the **Status Line.**

As the animation takes place any variables that become bound are changed on the screen from the variable name to the value they hold, along with a commentry on the binding. Similarly other features of Prolog such as backtracking and all the detail that backtracking causes are shown in detail with a commentry.

## Editor Window

The **Editor Window** contains the Prolog program or database. When the program is animated in the **Prolog Window**, the animated code is related to the program/database using highlighting. This window may scroll (move up and down) to bring the relevant Prolog clauses onto the screen.

## Status Line

The information contained in the Status Line continually changes, and tells a story of what is happening throughout the program animation, so keep an eye on it to see what is happening during a program animation.

## Prolog Syntax

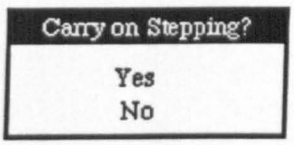
The syntax used in this Prolog is almost the same as you used at Summer School. If you remember a rule looked something like this:

```
rulehead(_X _Y) if
        subgoal1(_X) &
        subgoal2(_Y).
```

the only difference between the above syntax and the new syntax is the use of **'and'** instead of **'&'**. So the above rule will now look like this:

```
rulehead(_X _Y) if
        subgoal1(_X) and
        subgoal2(_Y).
```

One last thing is how you control the animation of the program. When the animation has commenced the display will stop at each important point in the program. When the animation stops, a menu will appear in the Prolog Window and will look like the picture below




**This is the menu that controls the animation**

The highlighted question on the menu says **'Carry on Stepping?'**. Underneath this there are two options **'Yes'** and **'No'**. If you choose **'Yes'** (which you should do) then the animation will proceed to the next stopping place. If you choose **'No'** the animation will stop and the query will be answered in the normal way. The way you make the choice of menu item is to move the mouse pointer over them chosen item, and to click (press) the left-hand mouse button.

When you select the **'Yes'** option from the menu the animation moves onto the next stopping place. However you should note that sometimes when the **'Yes'** option is selected the main display in the **Editor** and **Prolog** windows will not change, this is because the **Status Line** message has changed telling you about some forthcoming change in the display. So if you can't see a change in the display, have a look at the **Status Line** to see what the messsage is.

# APPENDIX-D

Appendix D presents the answers that subjects gave to the questionnaire in the misconceptions study presented in chapter 7. This study looked into the misconceptions novice programmers have about the workings of the Prolog interpreter.

The answers are grouped into misconception categories for each question/program in the questionnaire. Answers are presented for each program in the same order as they appear in the questionnaire, excluding the two model answers. The categories correspond to the categories shown in the analysis of the results in chapter 7, and along with them are shown the number of subjects that gave this answer, and the corresponding percentage.

**Program 1 'Warm-up'**

*Correct* 16 subjects (48.5%)

wine
food
_X = food

*Correct - no printing* 8 subjects (24.2%)

_X = food

*Static match* 3 subjects (9.1%)

food
wine
_X = food

*All subgoals must succeed /static match* 5 subjects (15.2%)

food
_X = food

*Miscellaneous* 1 subject (3.0%)

wine
food
food
mary
_X = food

**Program 2 'likes'**

*Correct* 23 subjects (69.7%)

fred
_X = jane _Y = fred
joe james
_X = june _Y = james

*Correct - no printing* 2 subjects (6.1%)

_X = jane _Y = fred
_X = june _Y = james

*All subgoals must succeed* 4 subjects (12.1%)
*/static match*

fred
_X = jane _Y = fred
james
_X = june _Y = james

*Miscallaneous* 4 subjects (12.1%)

1 subject (3.0%)
_X = jane _Y = fred

1 subject
fred
joe james
_X = june _Y = james

1 subject
_X = jane _Y = fred
joe

1 subject
fred
james

**Program 3 'connected'**

*Correct* 3 subjects (9.1%)

TWA194 PA100 AZ129 Paris Rome
_F1 = PA100 _F2 = AZ129

*Correct - no printing* 3 subjects (9.1%)

_F1 = PA100 _F2 = AZ129

*Static match* 16 subjects (51.5%)

TWA194 PA100 AZ129 Paris Rome Pisa
_F1 = PA100 _F2 = AZ129

*All subgoals must succeed /static match* 4 subjects (12.1%)

Rome
_F1 = PA100 _F2 = AZ129

*Real world fallacy* 2 subjects (6.1%)

Rome

*Miscellaneous* 4 subjects (12.1%)

1 subject (3.0%)
TWA194

1 subject (3.0%)
TWA194 PA100 AZ129 Chicago Dallas London

1 subject (3.0%)
TWA194 TWA194 PA100 PA100 AZ129 AZ129

1 subject (3.0%)
TWA194 Paris PA100 Rome
_F1 = PA100 _F2 = AZ129

**Program 4 'has_flu'**

*Correct - no printing* 3 subjects (9.1%)

yes

*Static match* 4 subjects (12.1%)

june

*Real world fallacy* 10 subjects (30.3%)

no

*Do not search from top* 4 subjects (12.1%)

john june yes 2 subjects

john june no 2 subjects

*Miscellaneous* 12 subjects (36.4%)

john yes 4 subjects

john no 2 subjects

mary yes 1subject

mary no 2 subjects

mary john 1 subject

mary june 1 subject

no attempt made 1 subject

**Program 5 'sisters1'**

*Do not search from top* 7 subjects (22.1%)

_X = alice _Y = june
_X = june _Y = alice

*Real world fallacy* 24 subjects (66.7%)

_X = alice _Y = june

_X = alice _Y = june 1 subject
_X = alice _Y = alice
_X = june _Y = june

_X = alice _Y = june 1 subject
_X = alice _Y = alice
_X = june _Y = june
_X = mary _Y = mary

*Miscellaneous* 2 subject (6.1%)

_X = alice _Y = alice 1 subject
_X = alice _Y = june

no attempt made 1 subject

**Program 6 'sisters2'**

| | |
|---|---|
| *Correct* | 20 subjects (60.6%) |
| yes | |
| *Miscellaneous* | 13 subjects (39.4%) |
| no | |

**Program 7 'abstract1'**

*Correct* 10 subjects (30.3%)

john mary jim sue
john mary jim sue
_X = sue
yes

*Correct - no printing* 3 subjects (9.1%)

_X = sue
yes

*All subgoals must succeed /static match* 1 subject (3.0%)

jim sue
_X = sue
yes

*Only try the first rule* 4 subjects (12.1%)

john mary jim sue

*Miscellaneous* 14 subjects (42.4%)

john mary jim sue 4 subjects
john mary jim sue
_X = mary
yes

_X = mary 2 subjects

john mary 2 subjects
_X = mary

john mary 1 subject
_X = john

john mary jim sue 1 subject

john mary jim sue 1 subject
_X = sue

no attempt made 3 subjects

no 1 subject

**Program 8 'abstract2'**

*All subgoals must succeed /static match* 8 subjects (24.2%)

foo
yes

*No backtracking* 3 subjects (9.1%)

bar baz foo 2 subjects
yes

foo bar baz fez foo 1 subject
yes

*Miscellaneous* 22 subjects (66.7%)

foo bar baz gort fez 11 subjects
yes

foo fez 2 subjects
yes

bar baz bar gort foo fez 2 subjects
yes

no attempt made 3 subjects

no 4 subjects

## APPENDIX-E

Appendix E presents the raw scores of the subjects who took part in the experiment that determined the effect of APT on novice programmers. The experiment is described in chapter 7, section 7.6.

The table below shows the raw scores for the APT group. Scores are given for each program in the questionnaires given to the subjects before and after they had seen the APT demonstrations showing the execution of the programs in the first questionnaire.

| Subject/Program | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| RP | before | 1 | 5 | 0 | 1 | 2 | 0 | 1 | 2 | 12 |
| | after | 4 | 3 | 0 | 3 | 2 | 0 | 0 | 1 | 13 |
| RM | before | 4 | 7 | 6 | 2 | 5 | 1 | 12 | 4 | 41 |
| | after | 1 | 8 | 7 | 3 | 5 | 1 | 12 | 9 | 46 |
| VC | before | 1 | 5 | 0 | 0 | 2 | 1 | 0 | 0 | 9 |
| | after | 1 | 10 | -2 | 3 | 7 | 1 | 7 | 1 | 28 |
| TJ | before | 1 | 5 | 2 | 0 | 1 | 0 | -2 | 2 | 9 |
| | after | 1 | 10 | 0 | 0 | 2 | 0 | 1 | 3 | 17 |
| PP | before | 3 | 10 | 10 | 3 | 2 | 0 | 5 | 8 | 41 |
| | after | 4 | 8 | 5 | 1 | 5 | 1 | 12 | 8 | 44 |
| DP | before | 4 | 9 | 4 | 1 | 5 | 0 | 3 | 8 | 31 |
| | after | 4 | 10 | 8 | 3 | 9 | 1 | 12 | 9 | 56 |
| Maximum score | | 4 | 10 | 11 | 5 | 14 | 1 | 12 | 9 | 67 |

The table below shows the raw scores for the 'control' group. Scores are given for each program in the questionnaires given to the subjects before and after they had seen the solutions to the programs in the first questionnaire.

| Subject/Program | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| LC | before | 4 | 3 | 1 | 1 | 0 | 0 | 5 | 1 | 16 |
| | after | 1 | 3 | 0 | 0 | 2 | 0 | 4 | 6 | 16 |
| MD | before | 4 | 2 | 5 | 2 | 2 | 0 | 2 | 1 | 18 |
| | after | 2 | 7 | 5 | 1 | 0 | 0 | 5 | 1 | 21 |
| NP | before | 2 | 8 | -2 | 2 | 2 | 0 | 0 | 1 | 13 |
| | after | -2 | 9 | 0 | 3 | 0 | 0 | 0 | 1 | 12 |
| TG | before | 4 | 8 | 0 | 3 | 2 | 0 | 0 | 0 | 17 |
| | after | 4 | 5 | 0 | 3 | 2 | 0 | 0 | 1 | 15 |
| DM | before | 1 | 10 | 1 | 0 | 2 | 0 | 1 | 0 | 15 |
| | after | 1 | 5 | 0 | 2 | 2 | 0 | 0 | 1 | 11 |
| CR | before | 1 | 7 | -2 | 0 | 2 | 1 | -1 | 1 | 9 |
| | after | 1 | 2 | 0 | 1 | 0 | 0 | 6 | 2 | 12 |
| Maximum score | | 4 | 10 | 11 | 5 | 14 | 1 | 12 | 9 | 67 |

## APPENDIX-F

Appendix F presents the questionnaire given to subjects in the explanations experiment. It consists of six Prolog programs, and questions asking the subjects to explain the meaning of particular rules.

## ***AN EXAMPLE OF HOW YOU SHOULD ANSWER THESE PROBLEMS***

*All answers are shown in italics*
ALL COMMENTS ARE SHOWN IN CAPITALS

```
likes(Tony _Y) if
   drinks(_Y wine) &
   PP(_Y) &
   tall(_Y).

drinks(Jon wine).
drinks(Hank wine).

tall(Hank).
```

---

Explain what the rule **'likes(_X _Y)'** means in English

EXPLANATION:
*The rule likes(Tony _Y) means that Tony likes somone _Y if that someone _Y drinks wine. and is tall*

ALTERNATIVE EXPLANATION:
*Tony likes all tall wine drinkers*

```
likes(mary  wine).
likes(mary  food).
likes(john  food).
likes(john  mary).

bothlike(_X)  if
   likes(mary  _X)  &
   PP(_X)  &
   likes(john  _X).
```

---

Explain what the rule '**bothlike(_X)**' means in English, ignoring the print statement '**PP(_X)**'.

```
has(fred  money).
has(joe  money).
has(james  money).

kisses(jane  fred).
kisses(june  james).

likes(_X  _Y)  if
   has(_Y  money)  &
   PP(_Y)  &
   kisses(_X  _Y).
```

---

Explain what the rule **'likes(_X _Y)'** means in English, ignoring the print statement **'PP(_Y)'**.

```
origin(BA137  Chicago).
origin(TWA194  Dallas).
origin(PA100  London).
origin(AZ129  London).

destination(TWA194  Paris).
destination(PA100  Rome).
destination(AZ129  Pisa).

stopover(BA137  Washington).
stopover(TWA194  Boston).
stopover(AZ129  Rome).

connected(_F1  _F2)  if
   destination(_F1  _X)  &
   PP(_F1)  &
   origin(_F2  _X).
connected(_F1  _F2)  if
   destination(_F1  _X)  &
   PP(_X)  &
   stopover(_F2  _X).
```

---

Explain what each rule **'connected(_F1 _F2)'** means in English, ignoring the print statements **'PP(_F1)'** and **'PP(_X)'**.

```
sisters(_X _Y) if
   female(_X) &
   parents(_X _M _F) &
   parents(_Y _M _F).

female(alice).
female(june).
female(mary).

parents(sue victoria fred).
parents(alice victoria albert).
parents(mary susan edward).
parents(june victoria albert).
parents(jenny jill roy).
```

---

Explain what the rule '**sisters(_X _Y)**' means in English.

```
a(_X) if
  b(_X) &
  c(_X).

b(_X) if
  h(_X _Y) &
  PP(_X _Y) &
  i(_Y).
b(_X) if
  h(_Y _X) &
  PP(_Y _X) &
  i(_Y).

h(1 2).
h(3 4).

i(2).
i(3).

c(2).
c(4).
```

---

Explain what each rule '**b(_X)**' means in English, ignoring the print statements '**PP(_X _Y)**' and '**PP(_Y _X)**'.

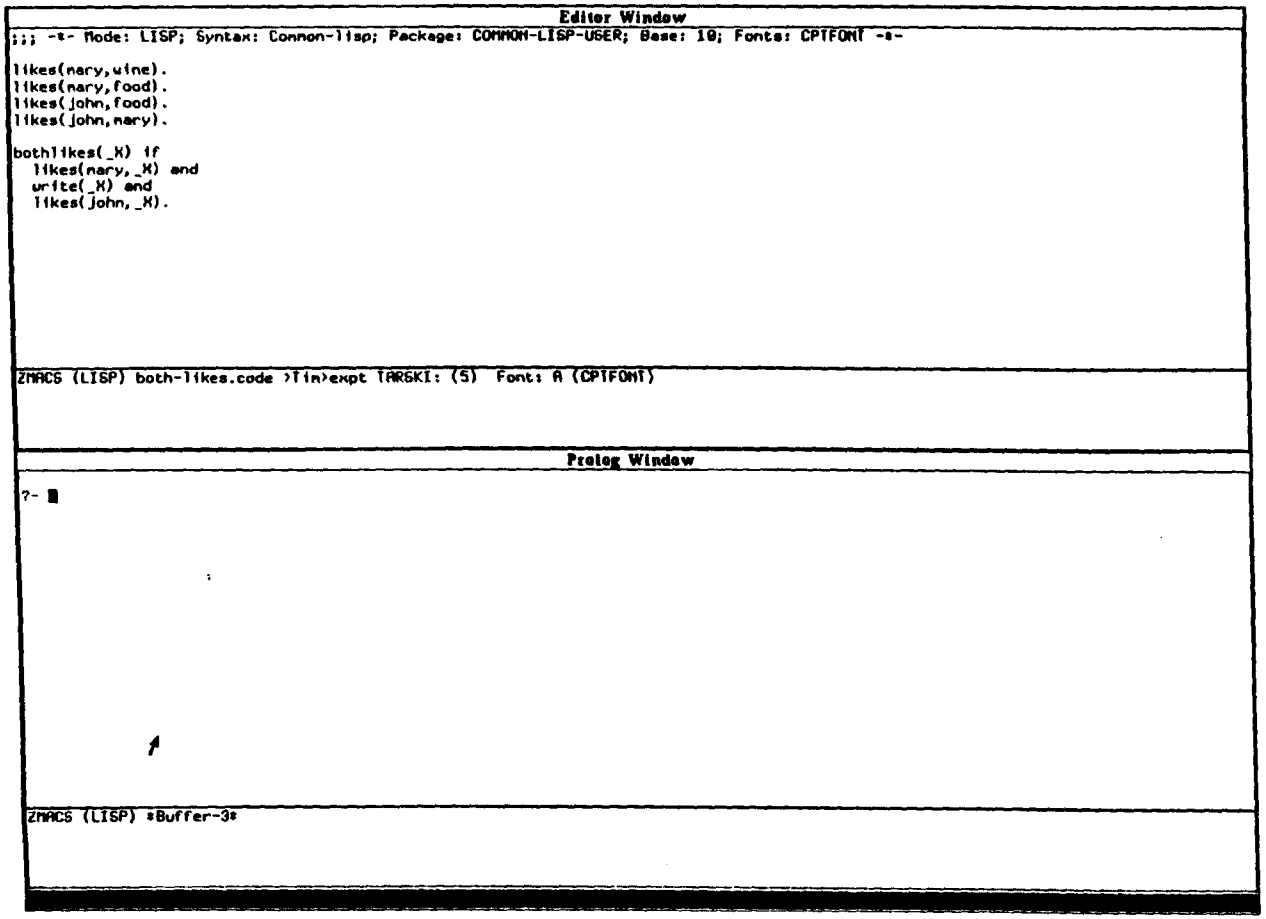
```
a if
  b &
  loves(John Mary).
a if c.

b if
  d &
  e.
b if f.

c if PP(foo).

d if PP(bar).

e if PP(baz).

e if PP(gort).

f if PP(fez).

loves(John Sue).
```

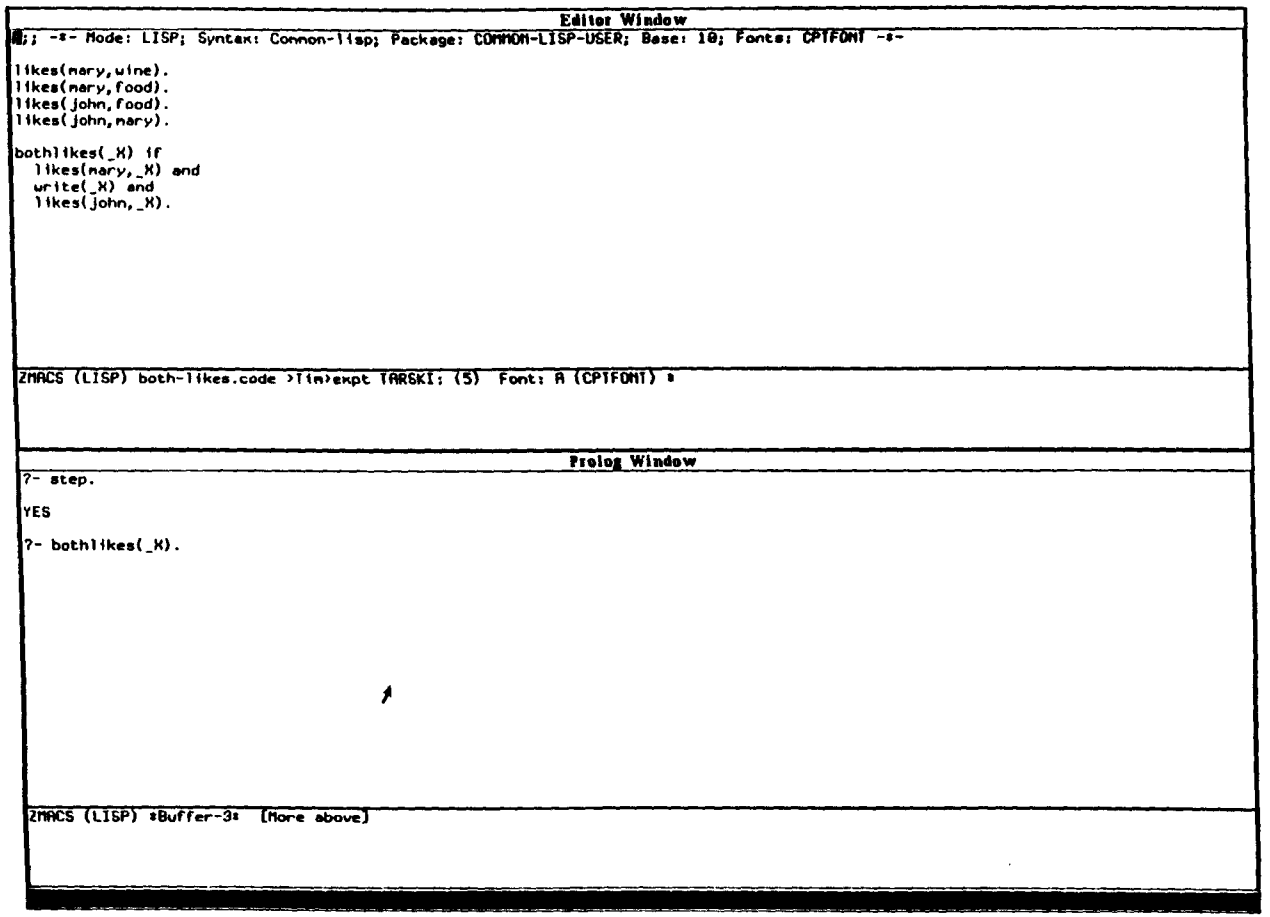
---

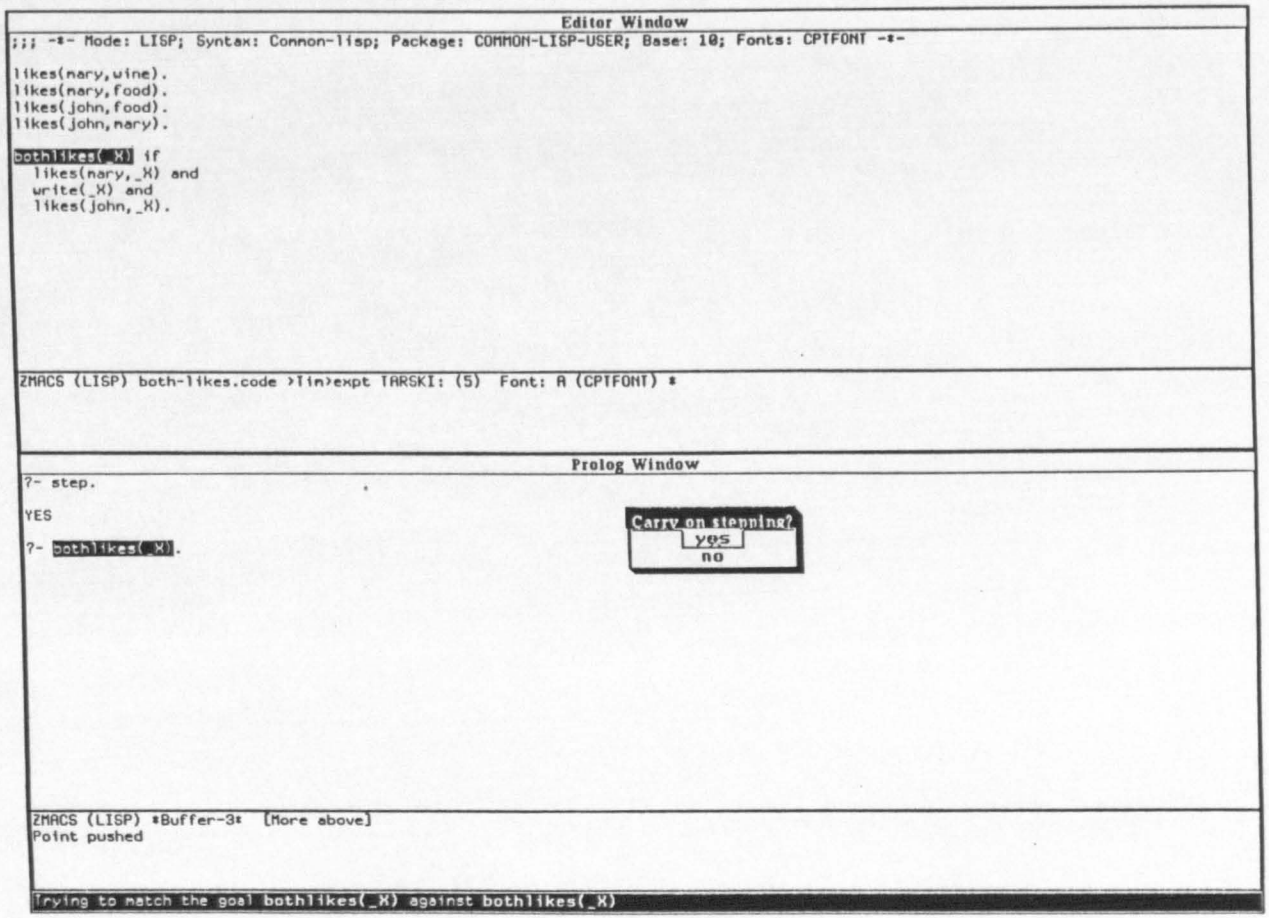
Explain what the rules '**a**' and '**b**' mean in English.

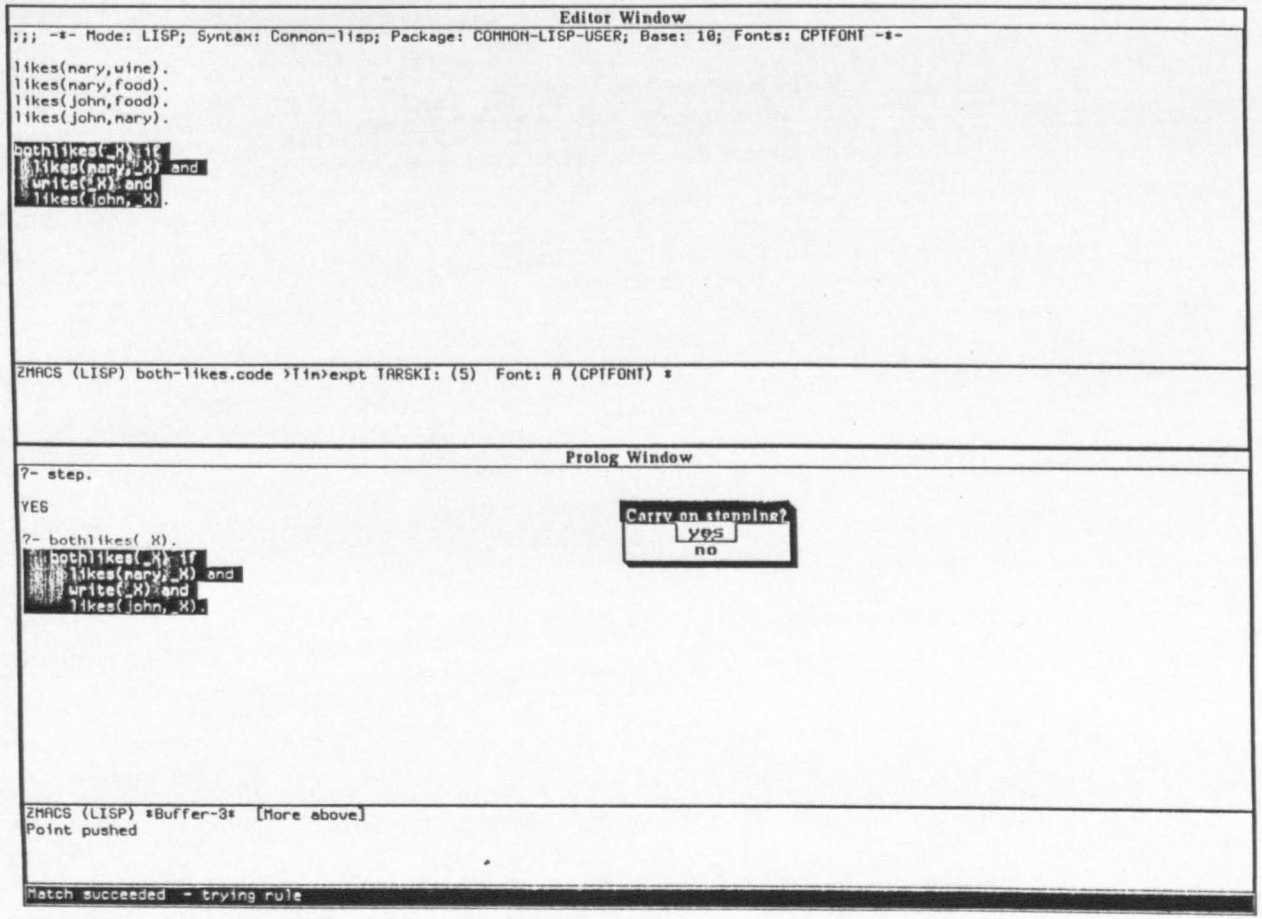
## APPENDIX-G

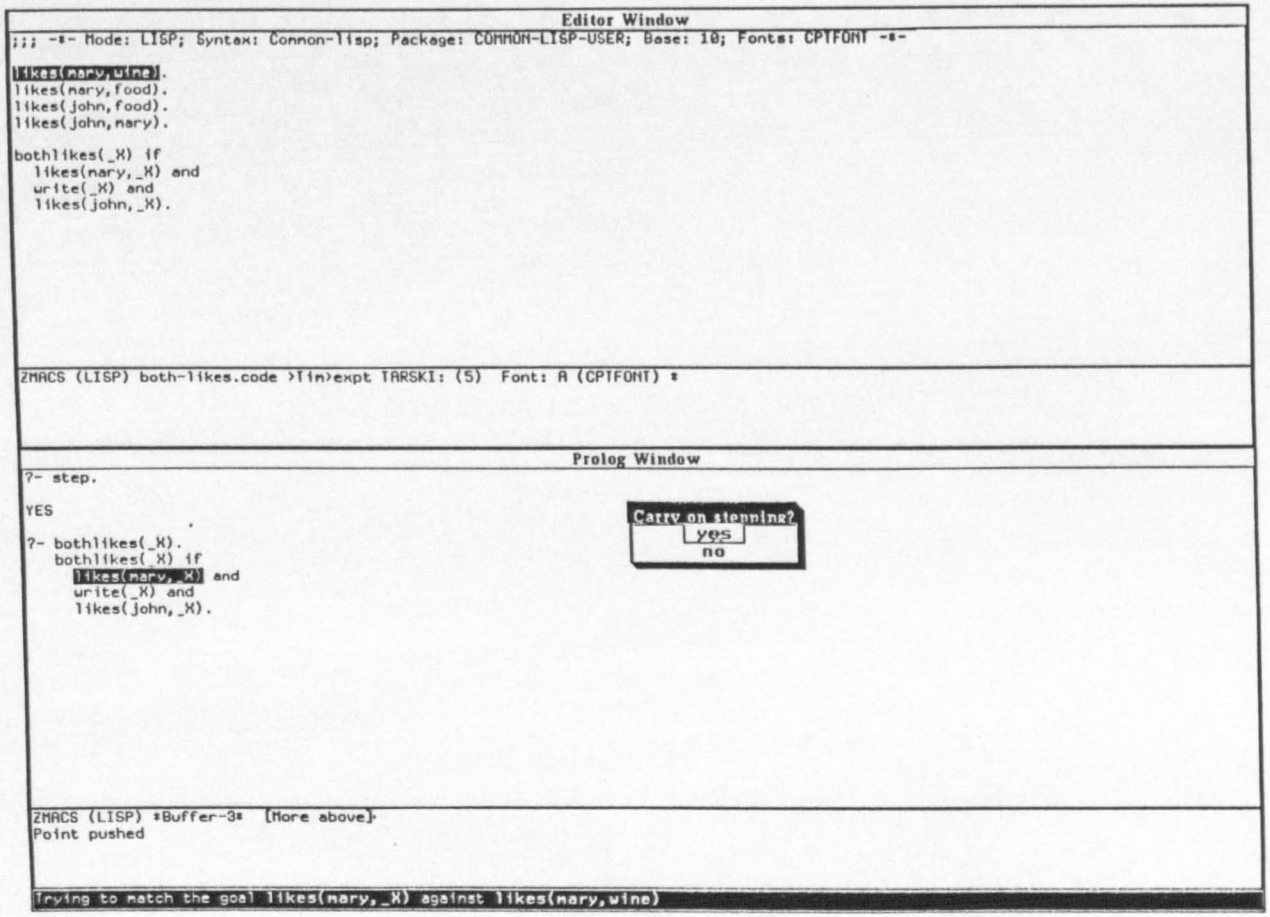
Appendix G presents a complete listing of the execution of a Prolog program as displayed by APT. The listing consists of a series of screen snapshots which have been taken at each step in the program. The program used in this example is program 1 'Warm-up' which is taken from the questionnaire described in Appendix A.

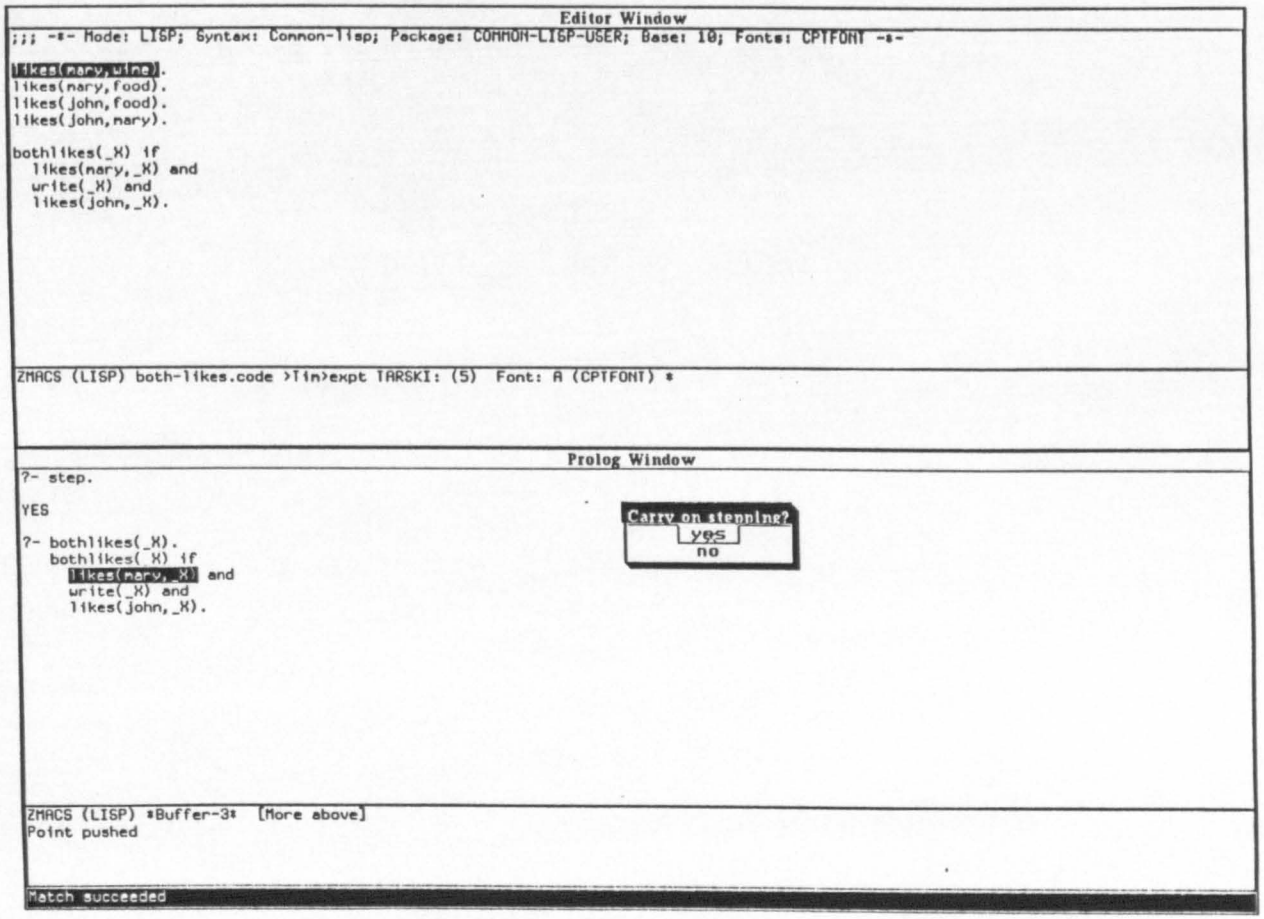

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT)
Prolog Window
?-
ZMACS (LISP) *Buffer-3*

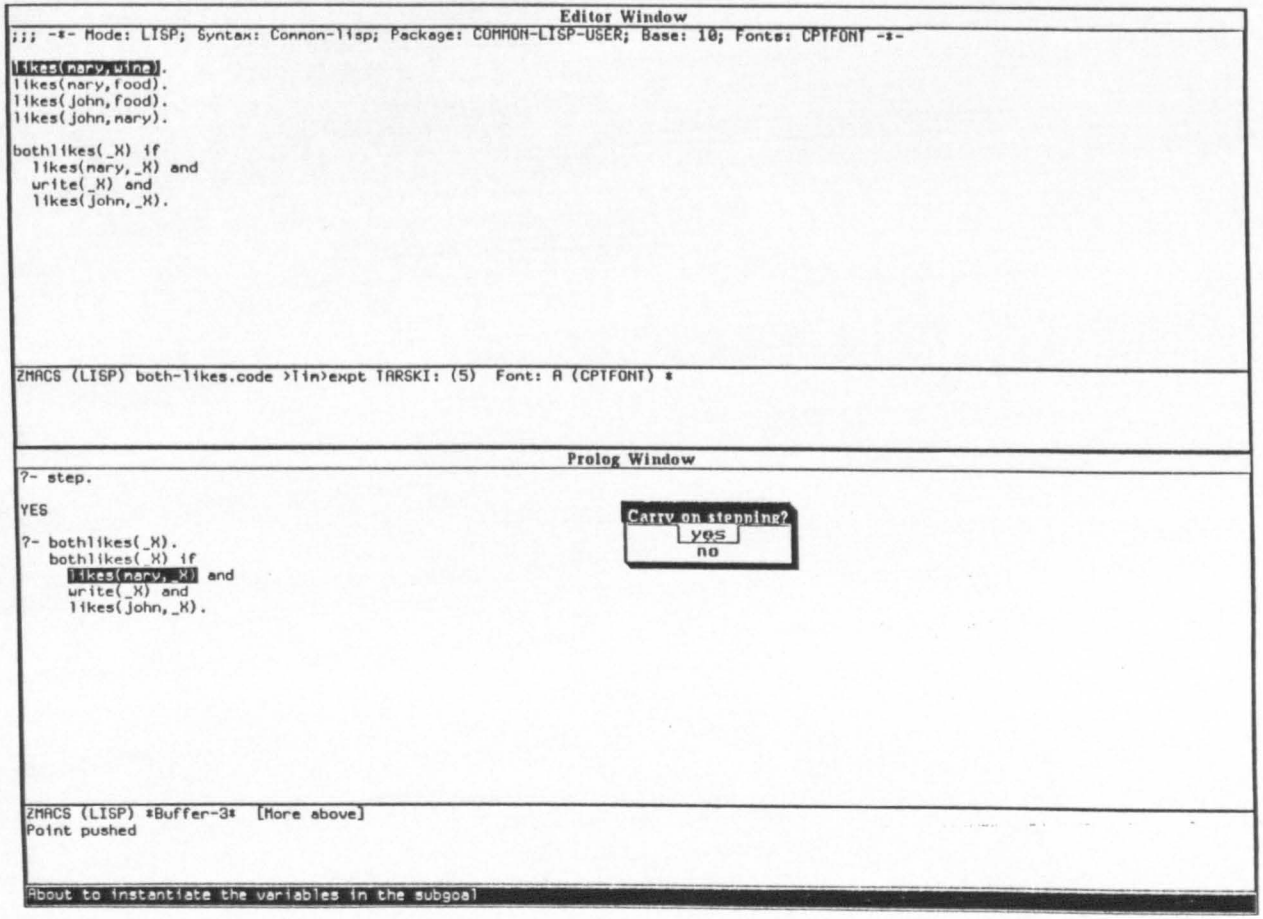

Editor Window
;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI; (5) Font: A (CPTFONT)
Prolog Window
?- step.
YES
?- bothlikes(_X).
ZMACS (LISP) *Buffer-3* [More above]

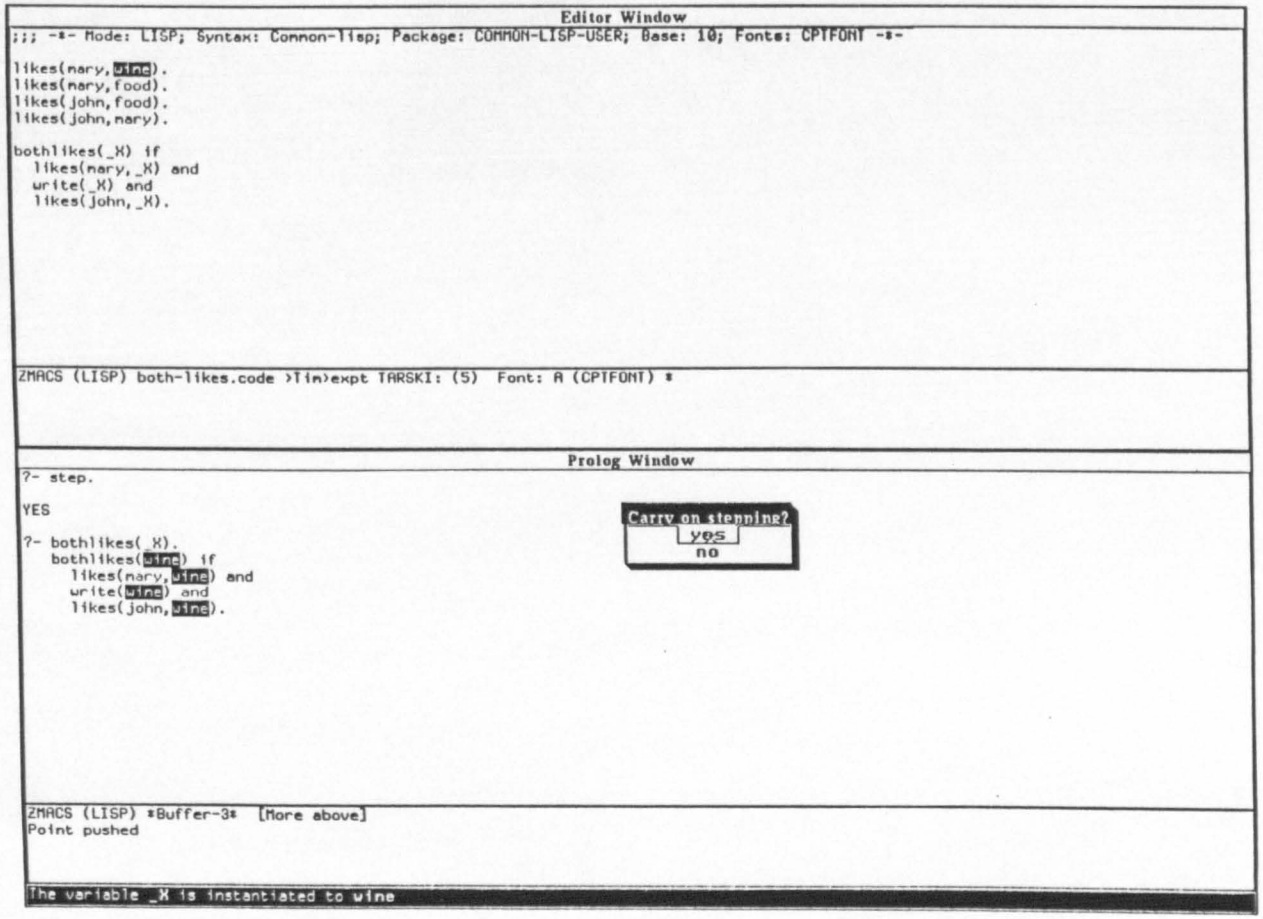

Editor Window
;;; -*- Mode: LISP; Syntax: Connon-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal bothlikes(_X) against bothlikes(_X)

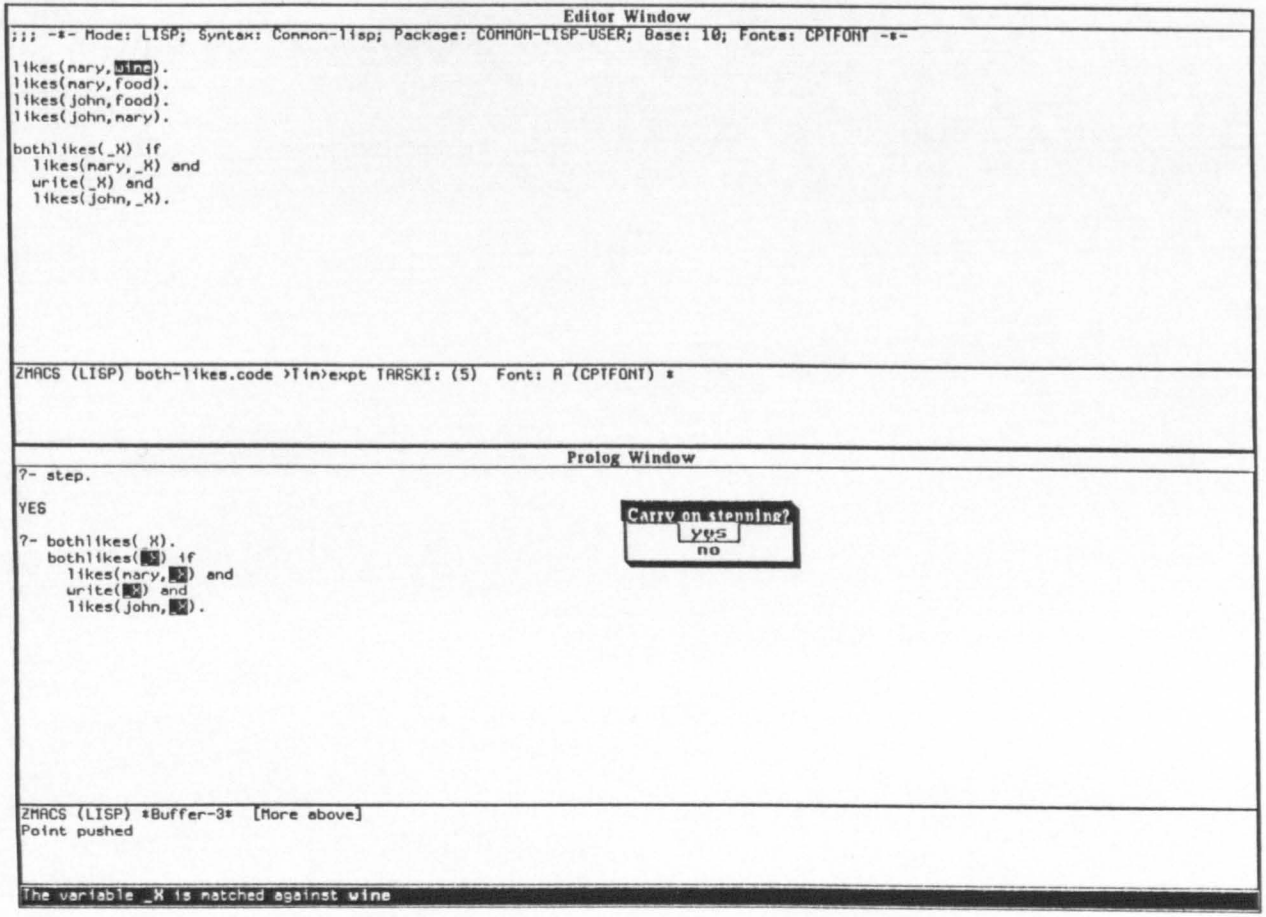

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X):-
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes( X).
bothlikes(_X):-
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded - trying rule

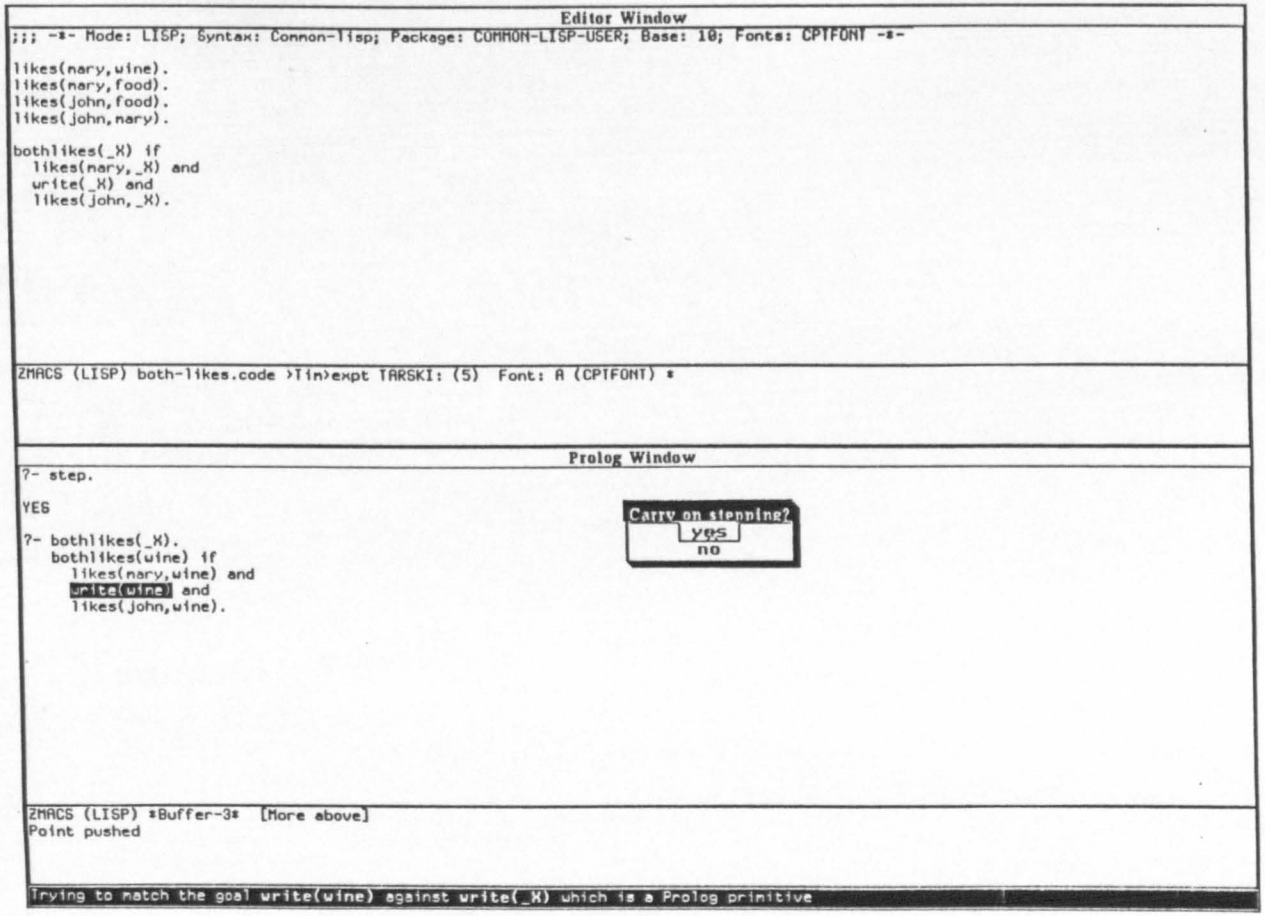

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Font: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(mary,_X) against likes(mary,wine)

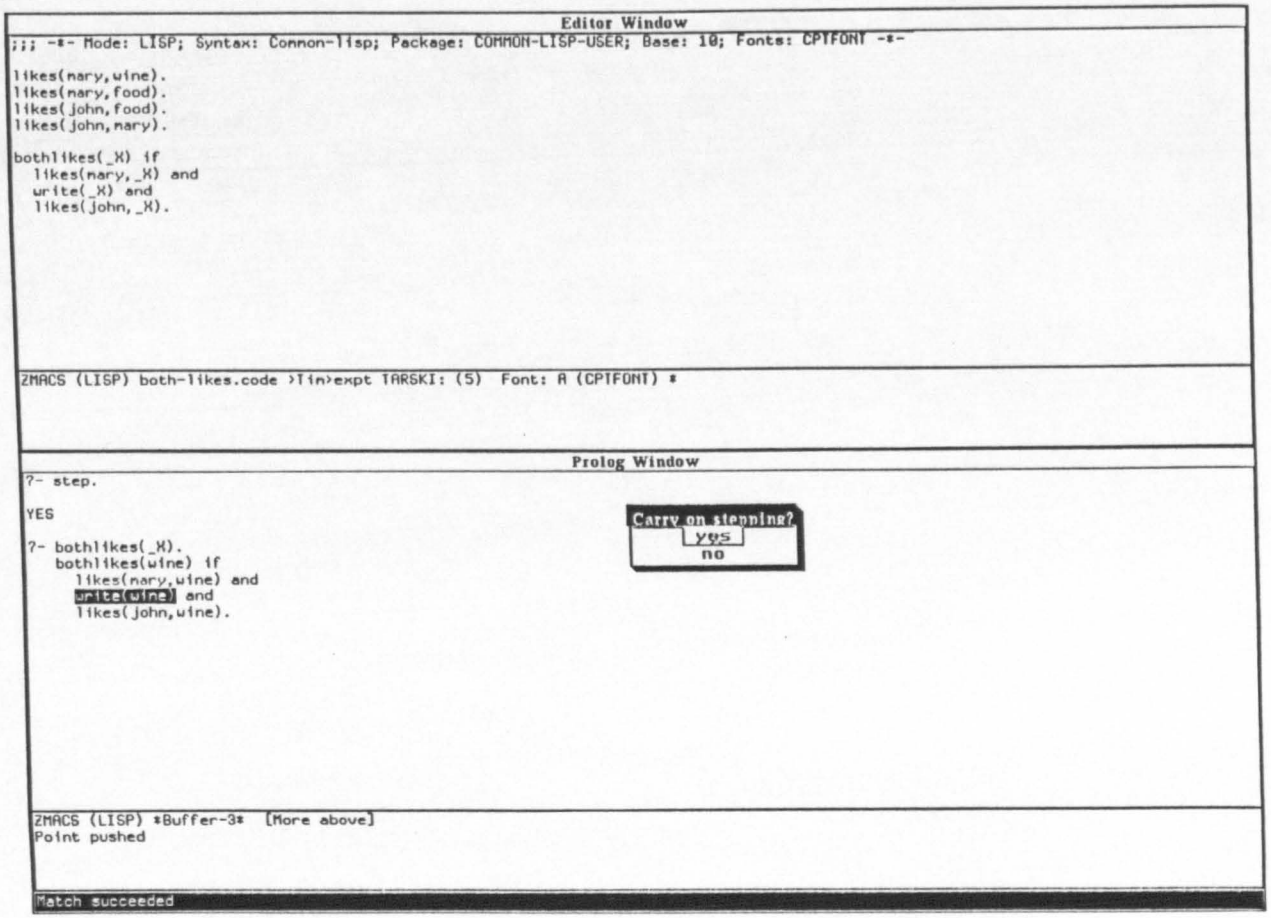

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded

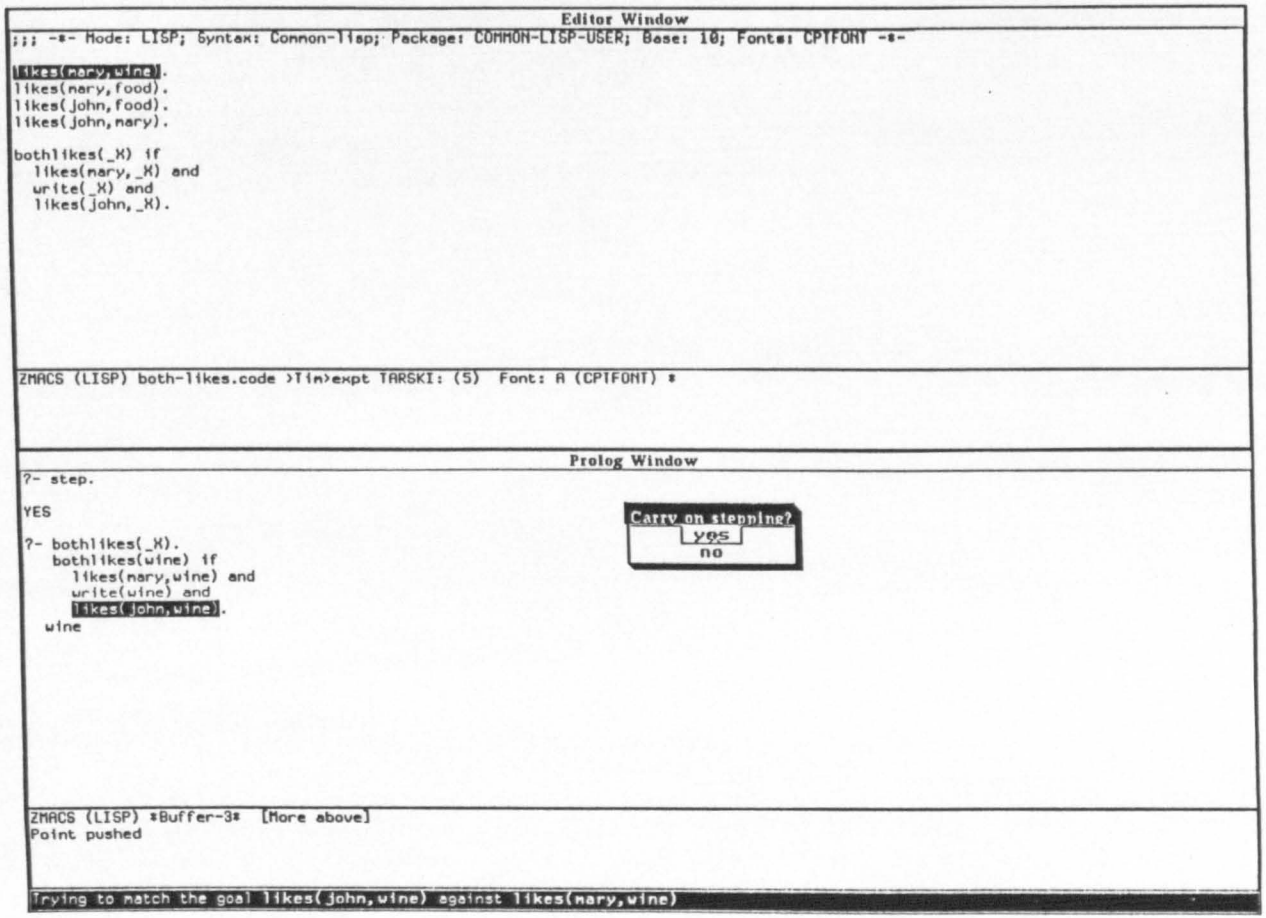

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
About to instantiate the variables in the subgoal

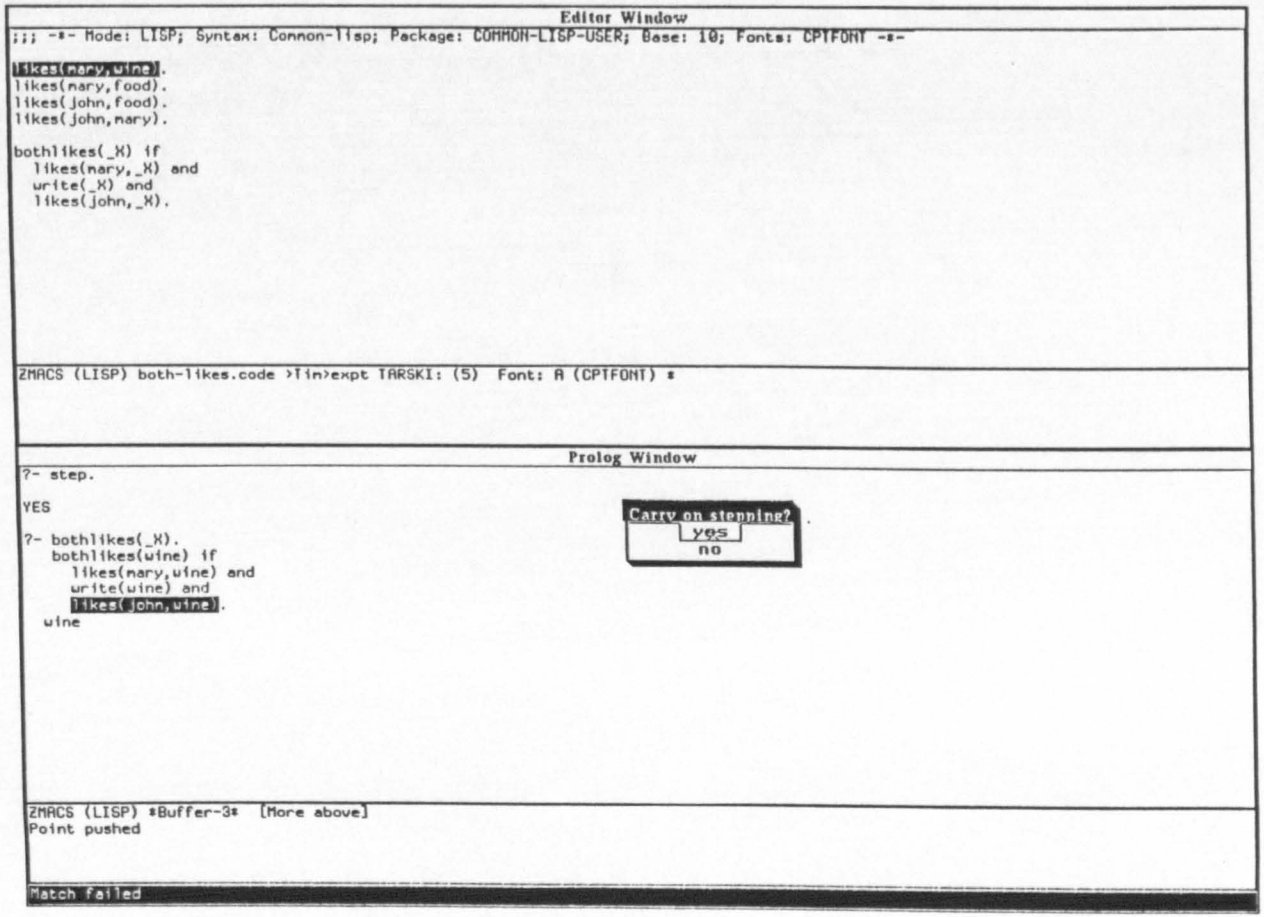

Editor Window
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _X is instantiated to wine

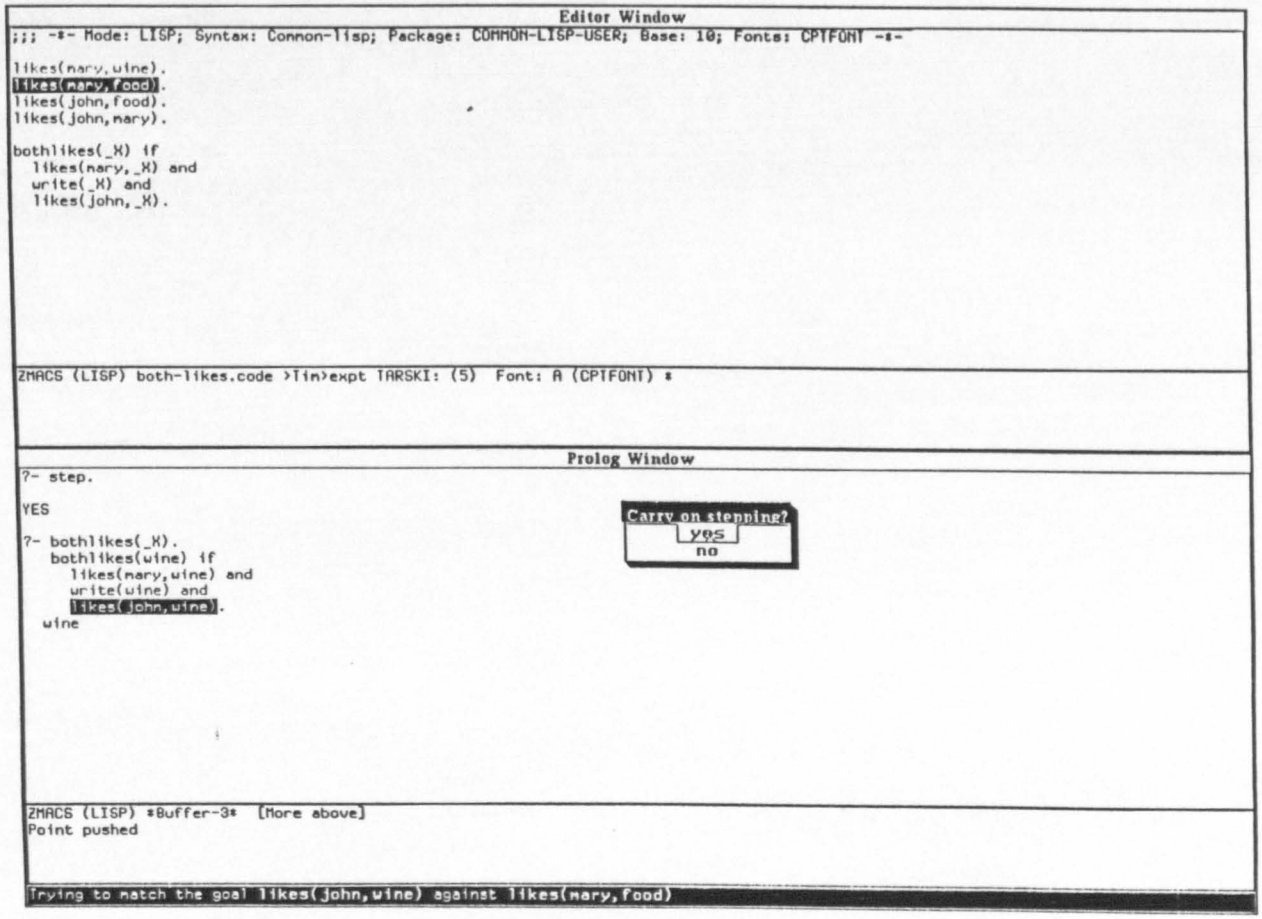

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _X is matched against wine

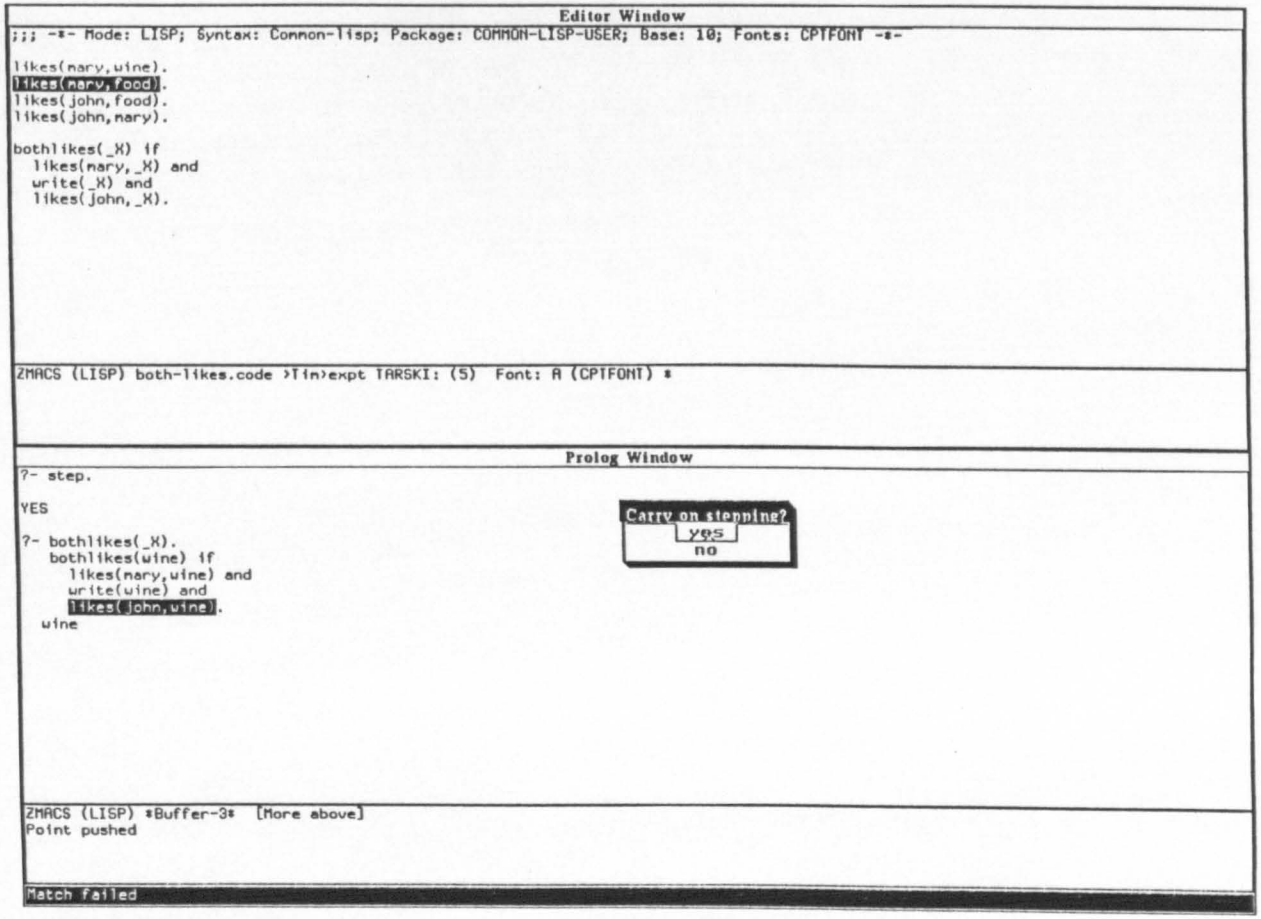

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(nary,wine).
likes(nary,food).
likes(john,food).
likes(john,nary).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(nary,wine) and
write(wine) and
likes(john,wine).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal write(wine) against write(_X) which is a Prolog primitive

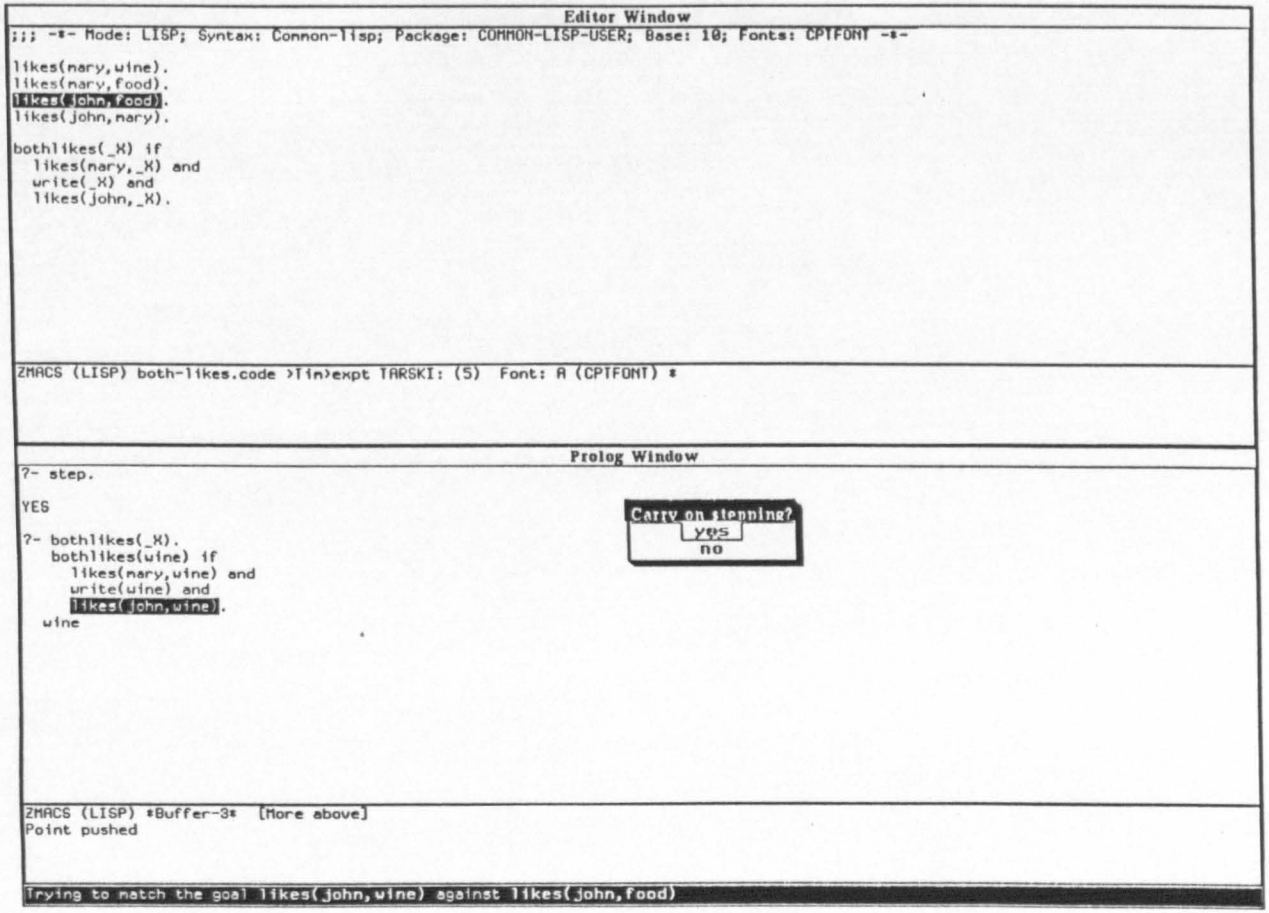

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >(in>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded

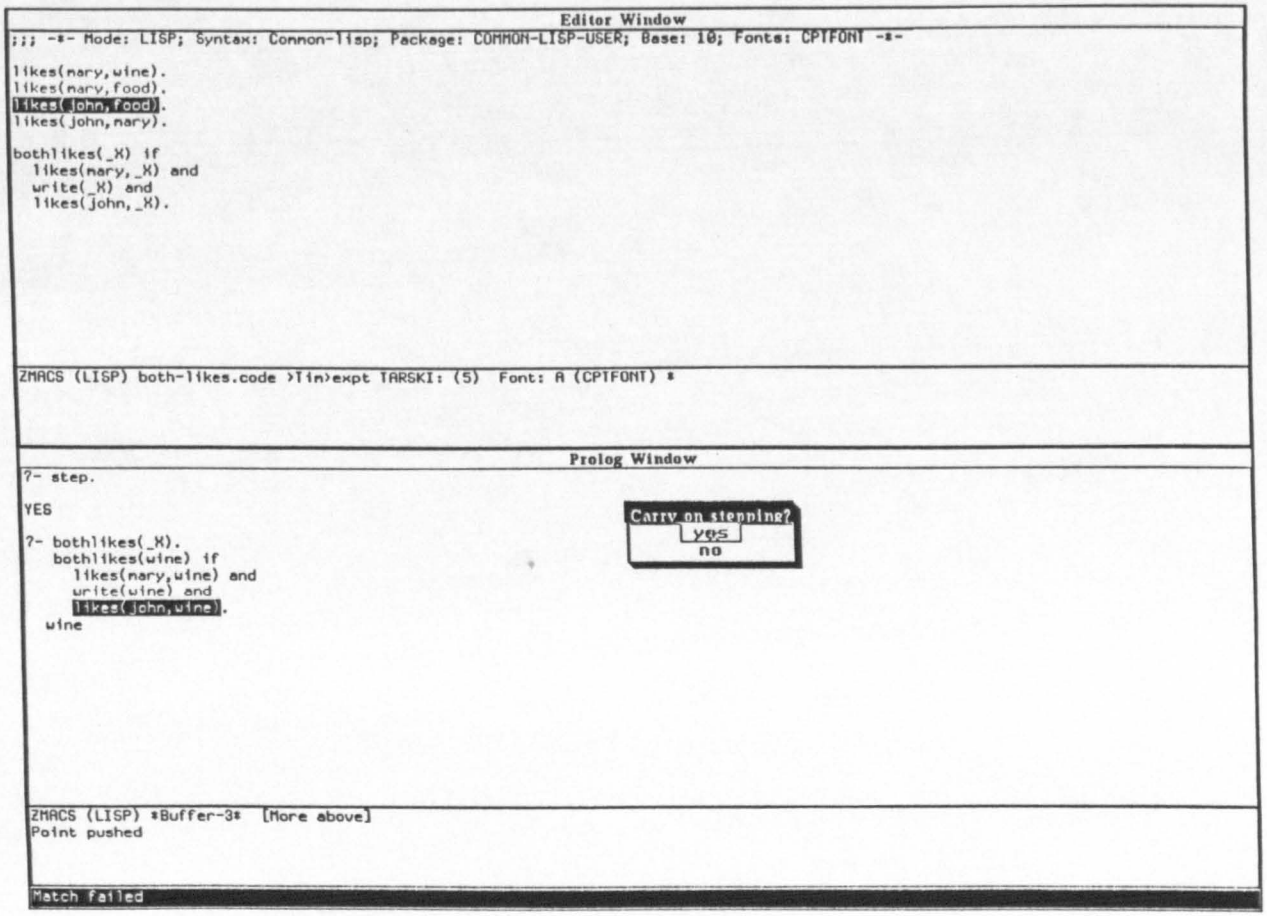

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Font: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(john,wine) against likes(mary,wine)

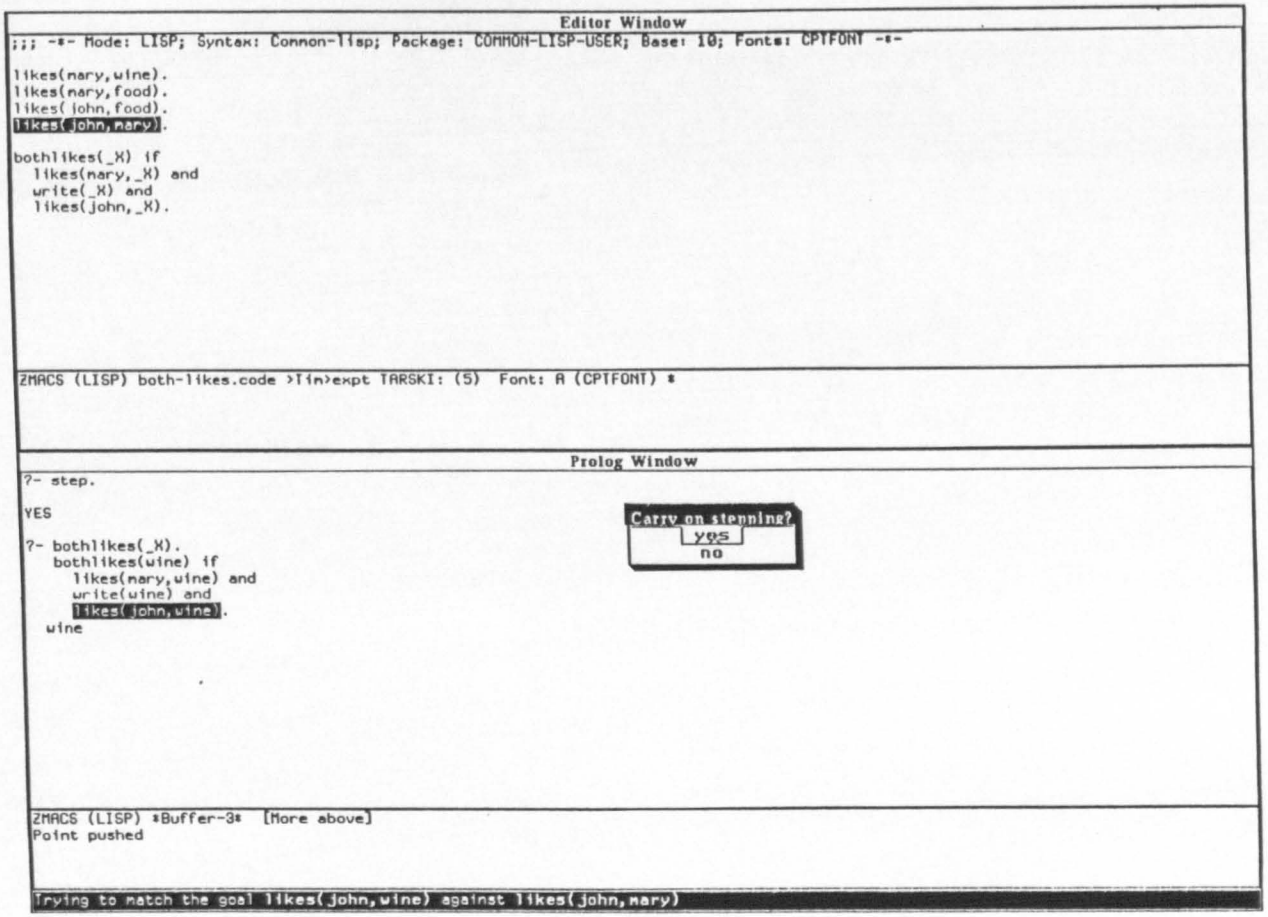

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed

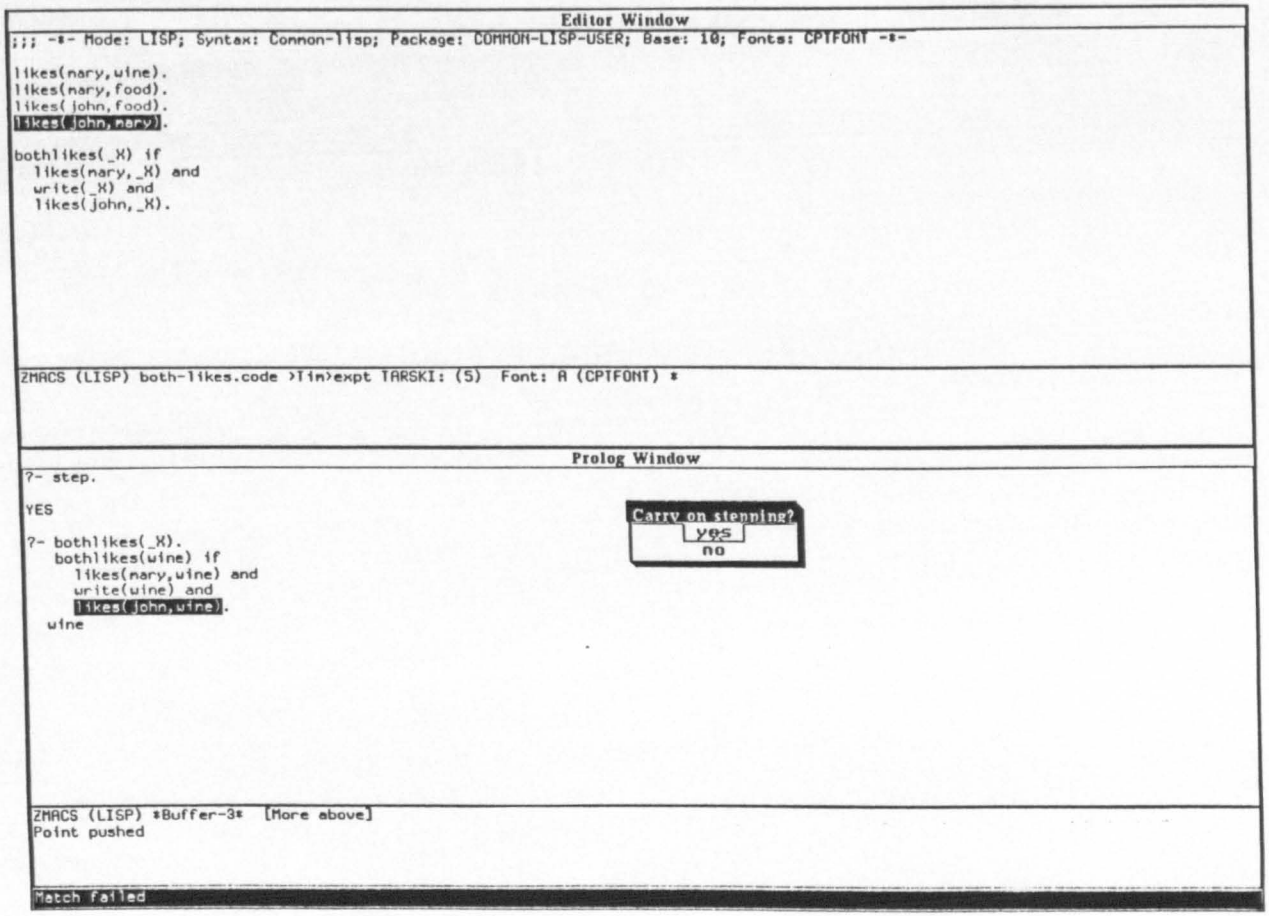

Editor Window
;;; -*- Mode: LISP; Syntax: Connon-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(john,wine) against likes(mary,food)

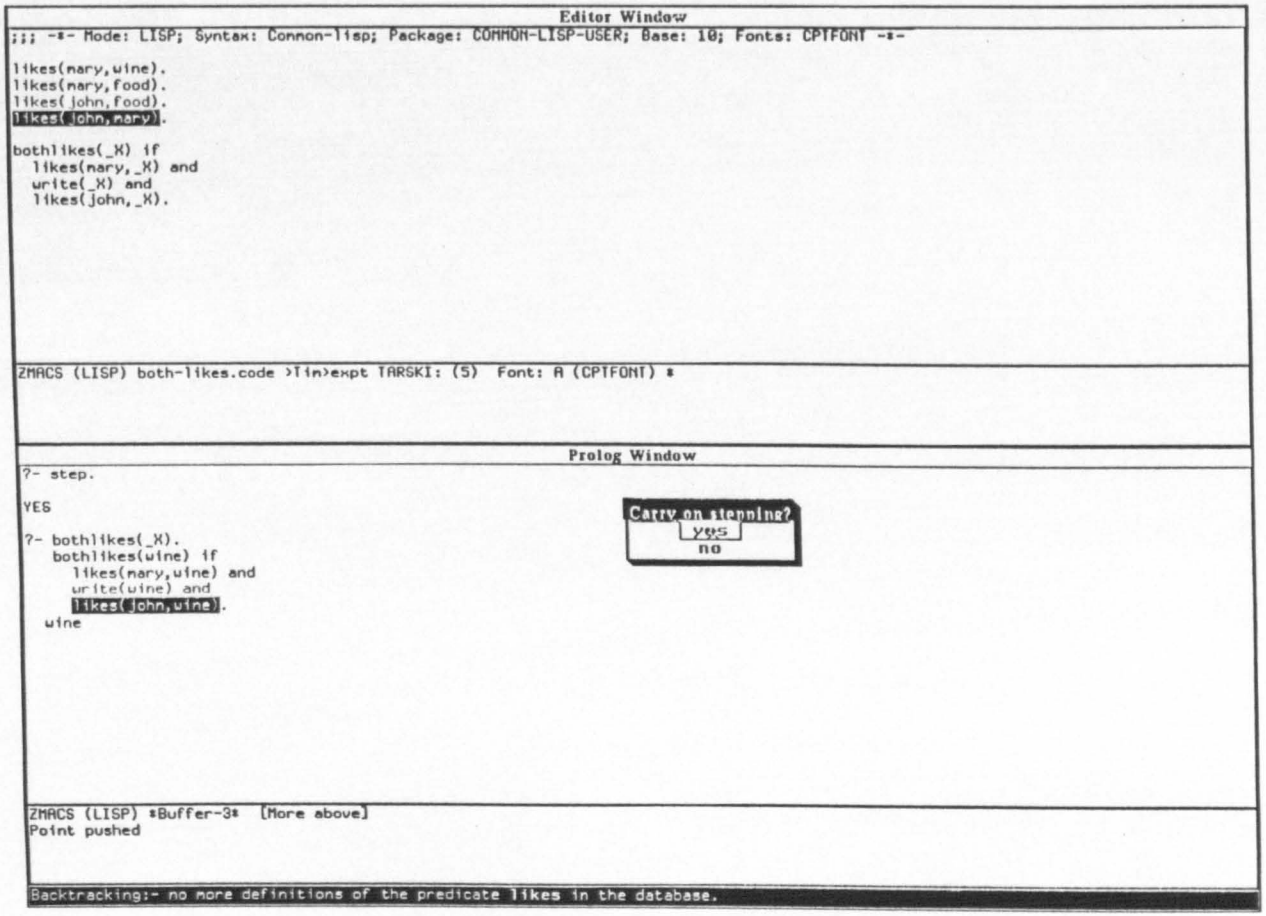

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed

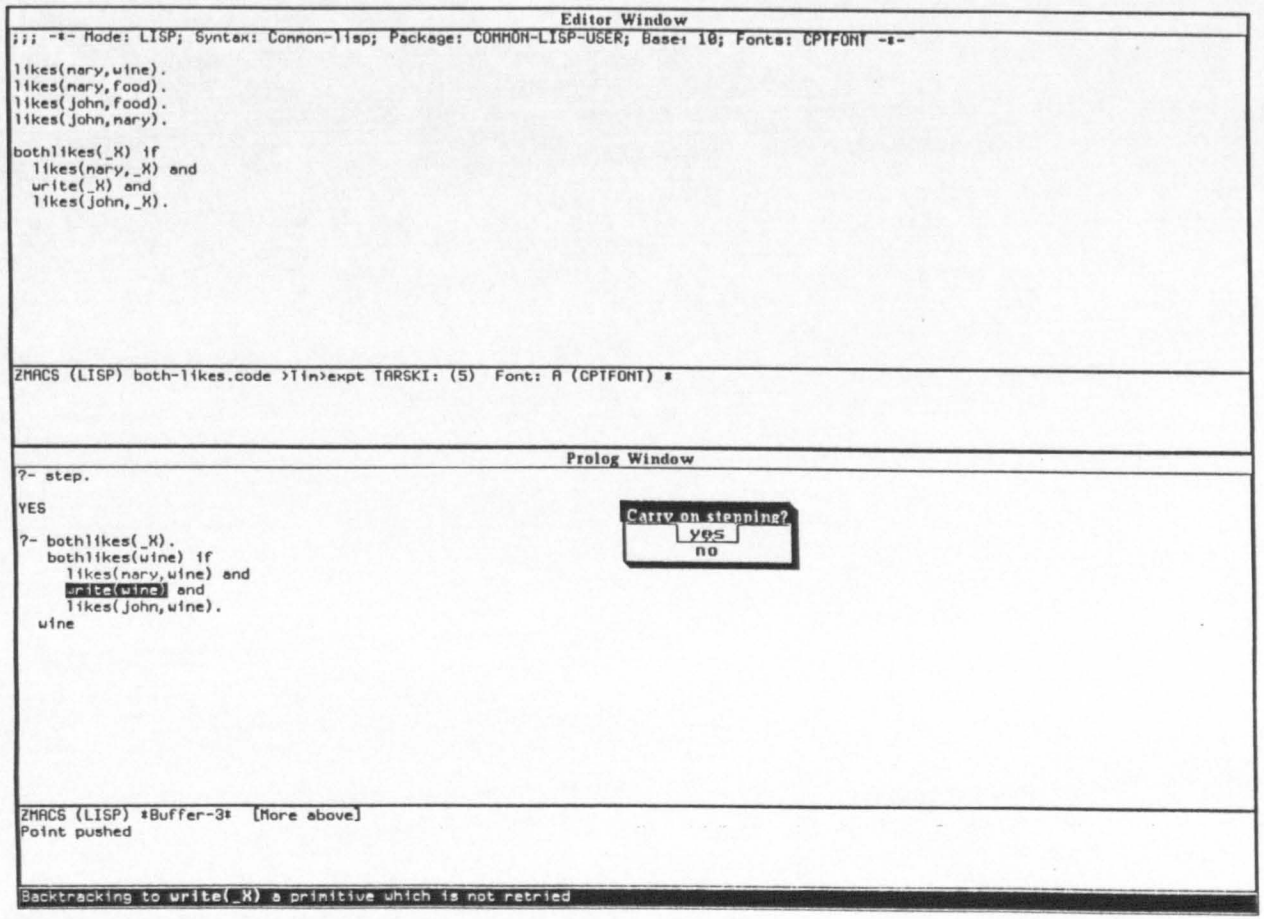

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(john,wine) against likes(john,food)

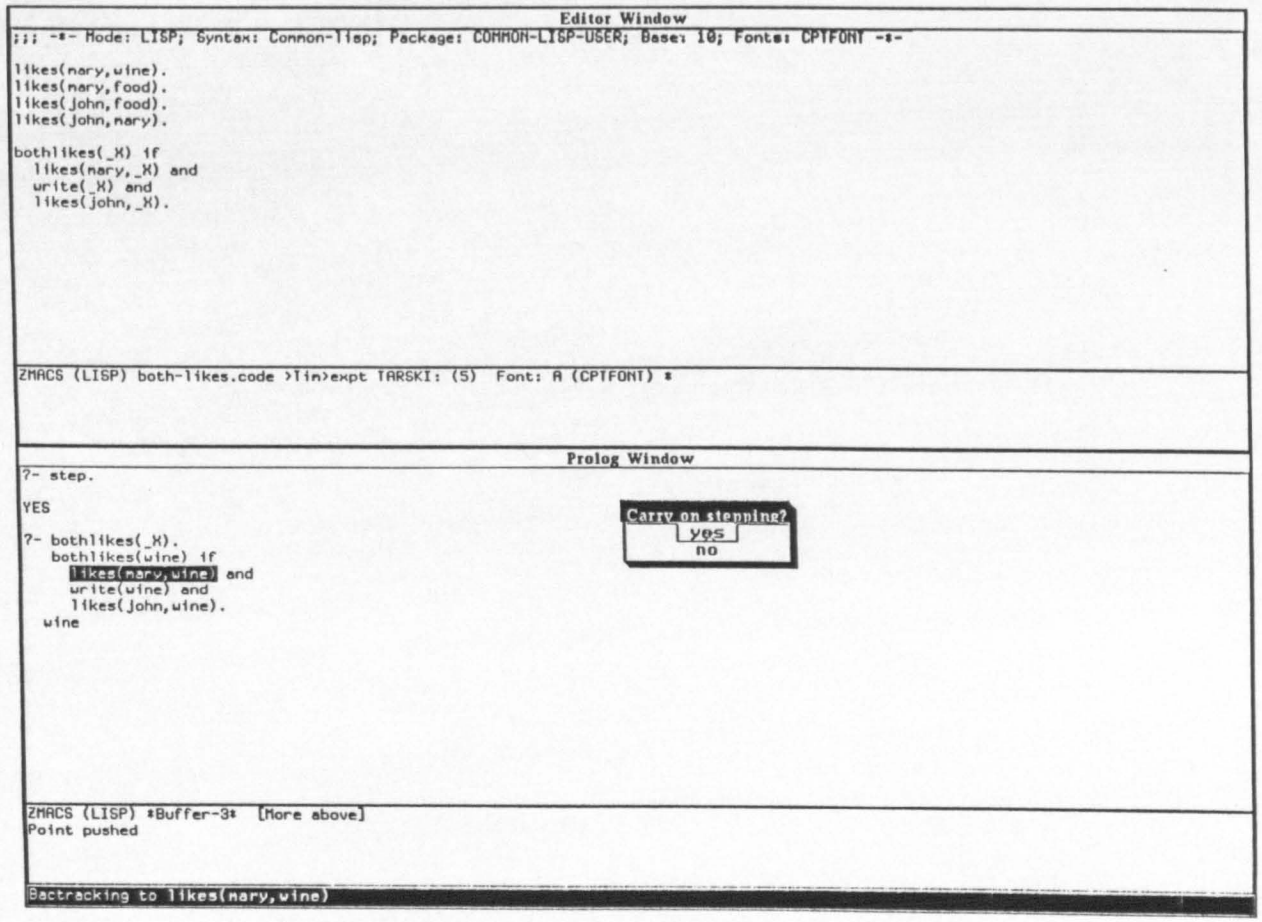

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed

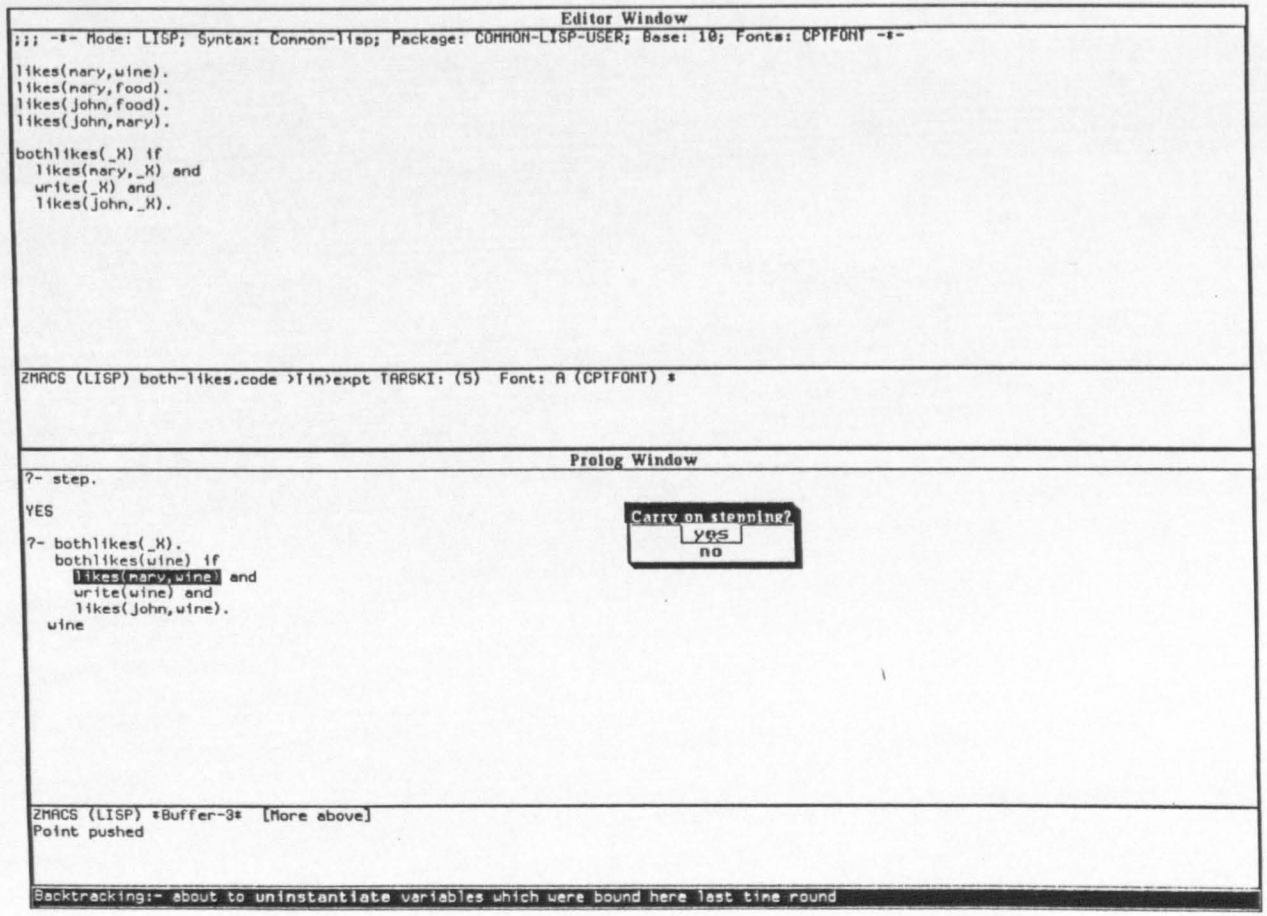

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Font*: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(john,wine) against likes(john,mary)

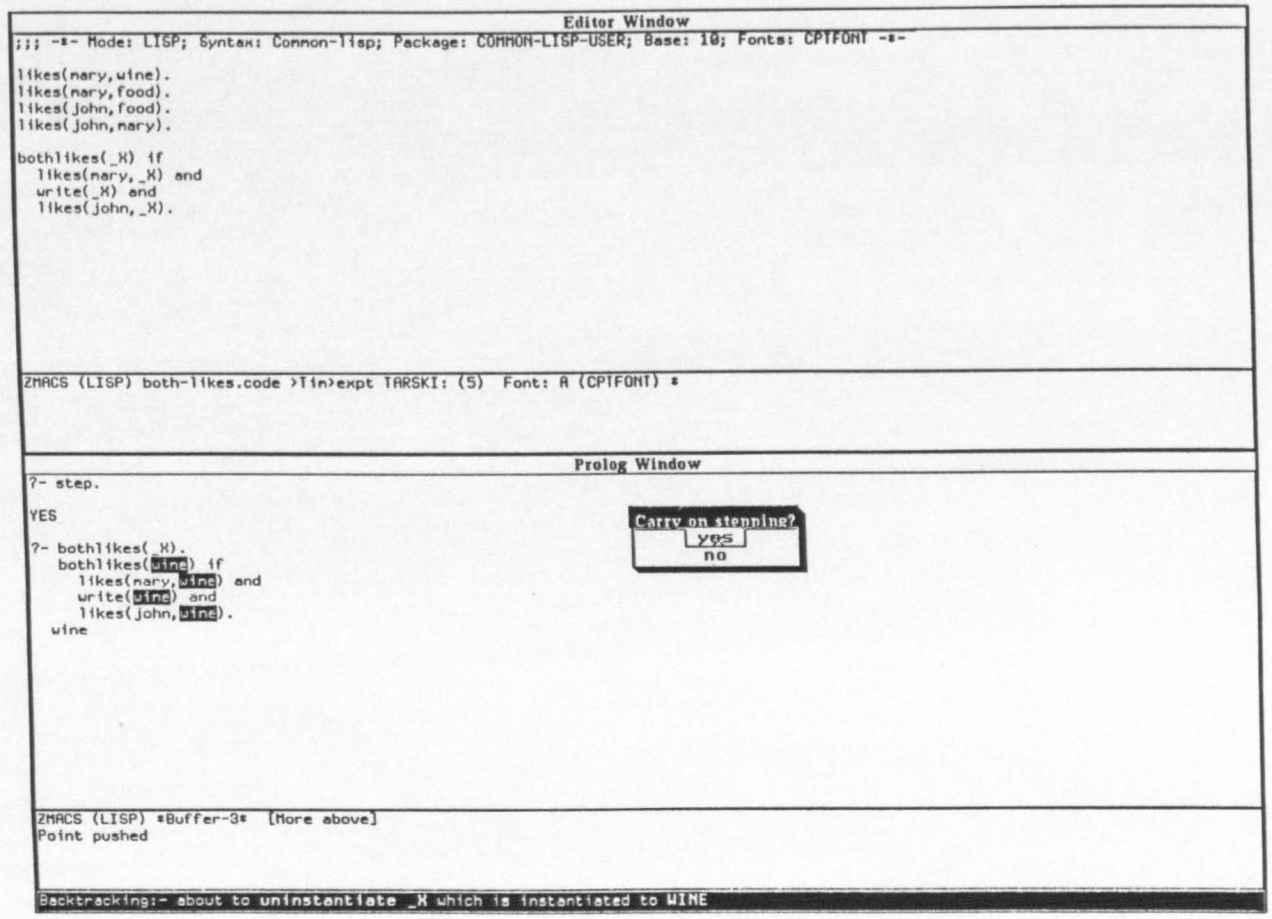

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(nary,wine).
likes(nary,food).
likes(john,food).
likes(john,nary).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(nary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed

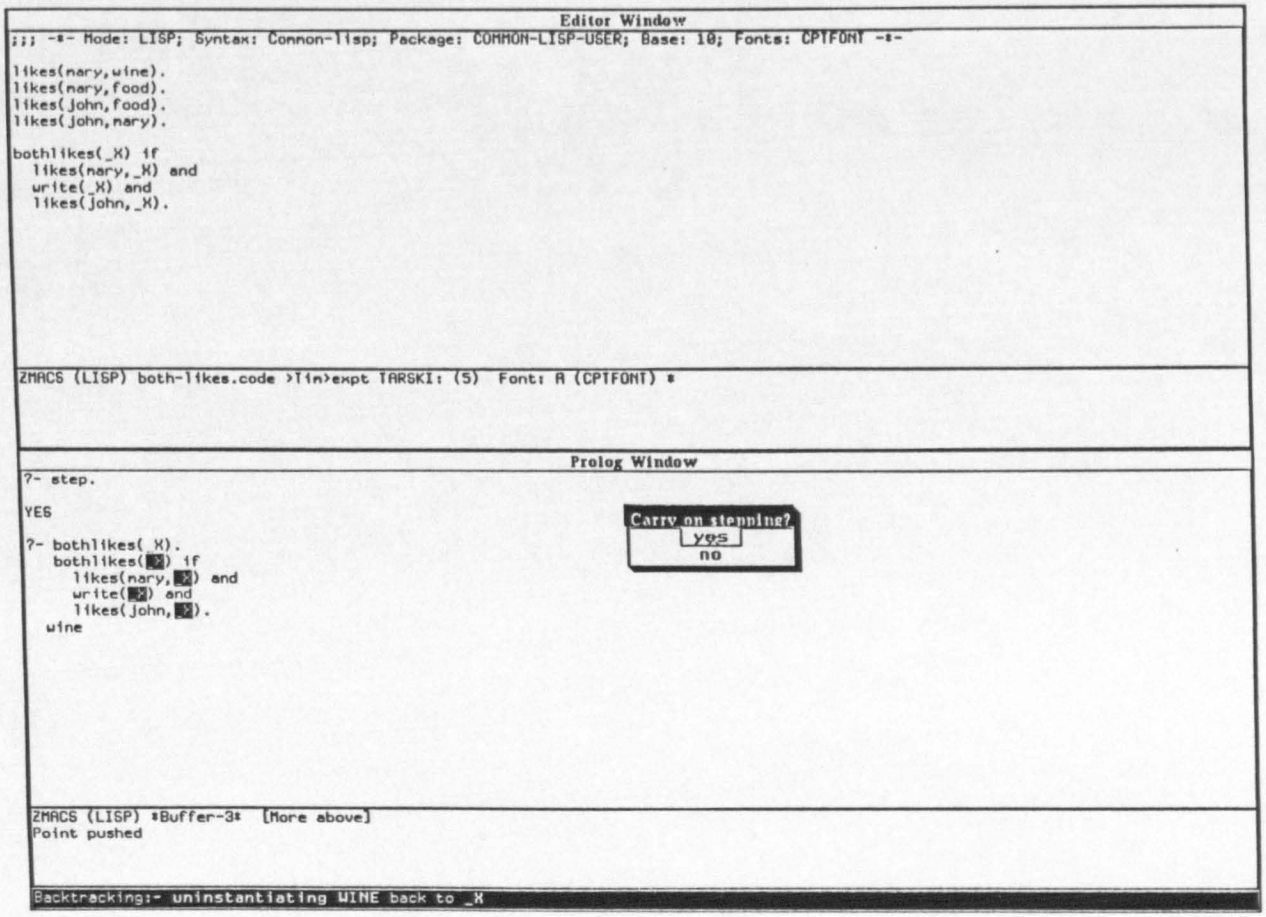

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(nary,wine).
likes(nary,food).
likes(john,food).
likes(john,nary).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
Carry on stepping?
yes
no
?- bothlikes(_X).
bothlikes(wine) if
likes(nary,wine) and
write(wine) and
likes(john,wine).
wine
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- no more definitions of the predicate likes in the database.

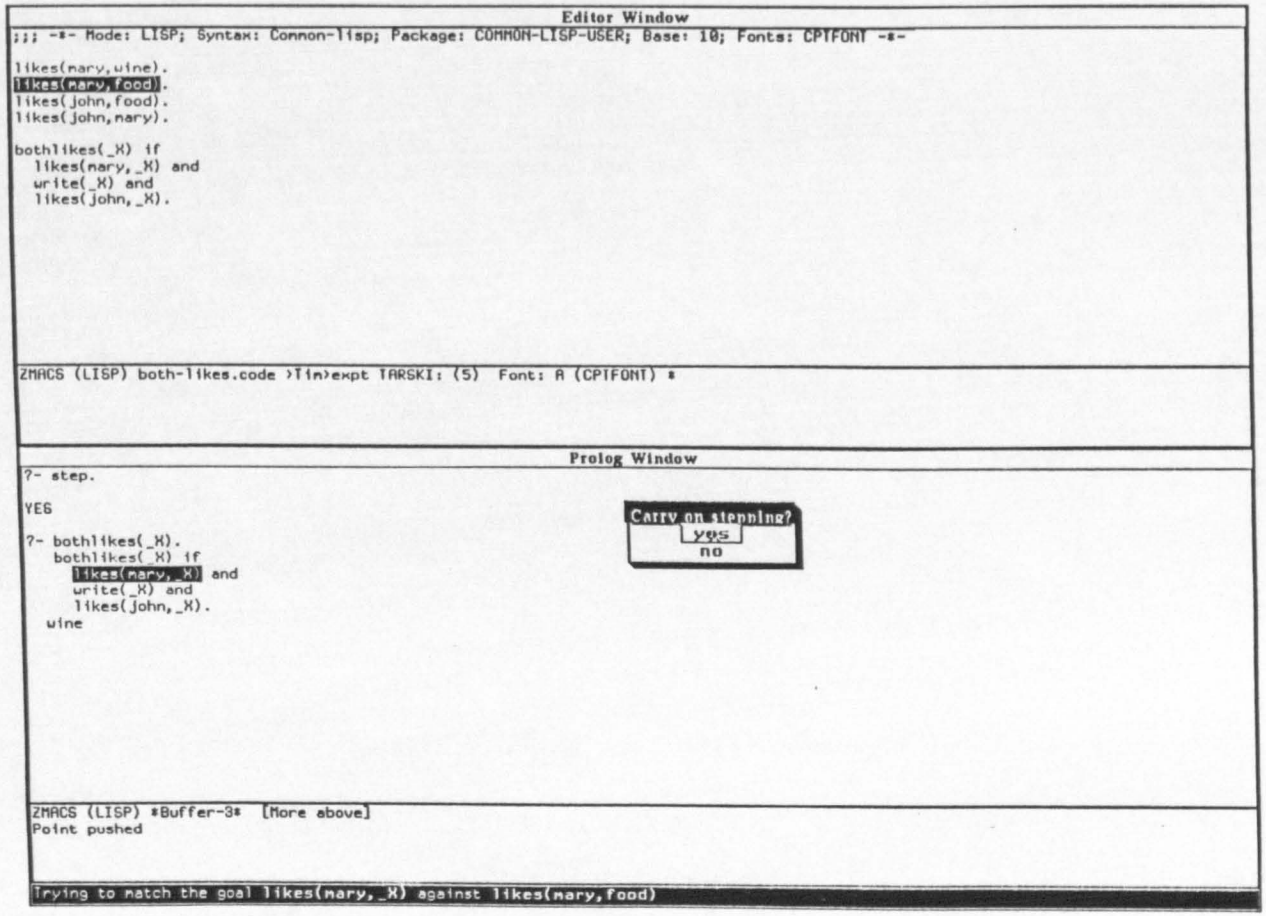

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking to write(_X) a primitive which is not retried

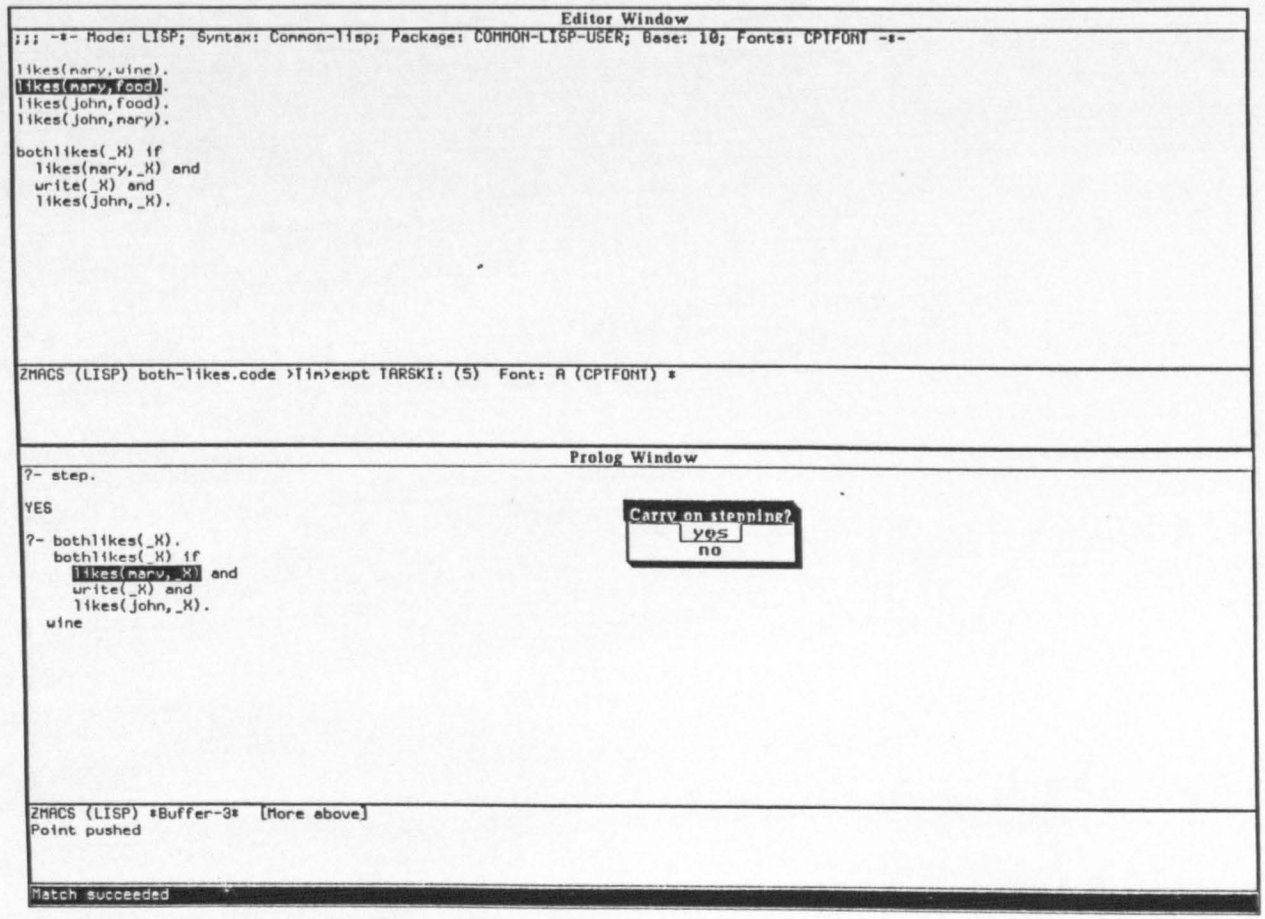

Editor Window
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(mary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking to likes(mary,wine)

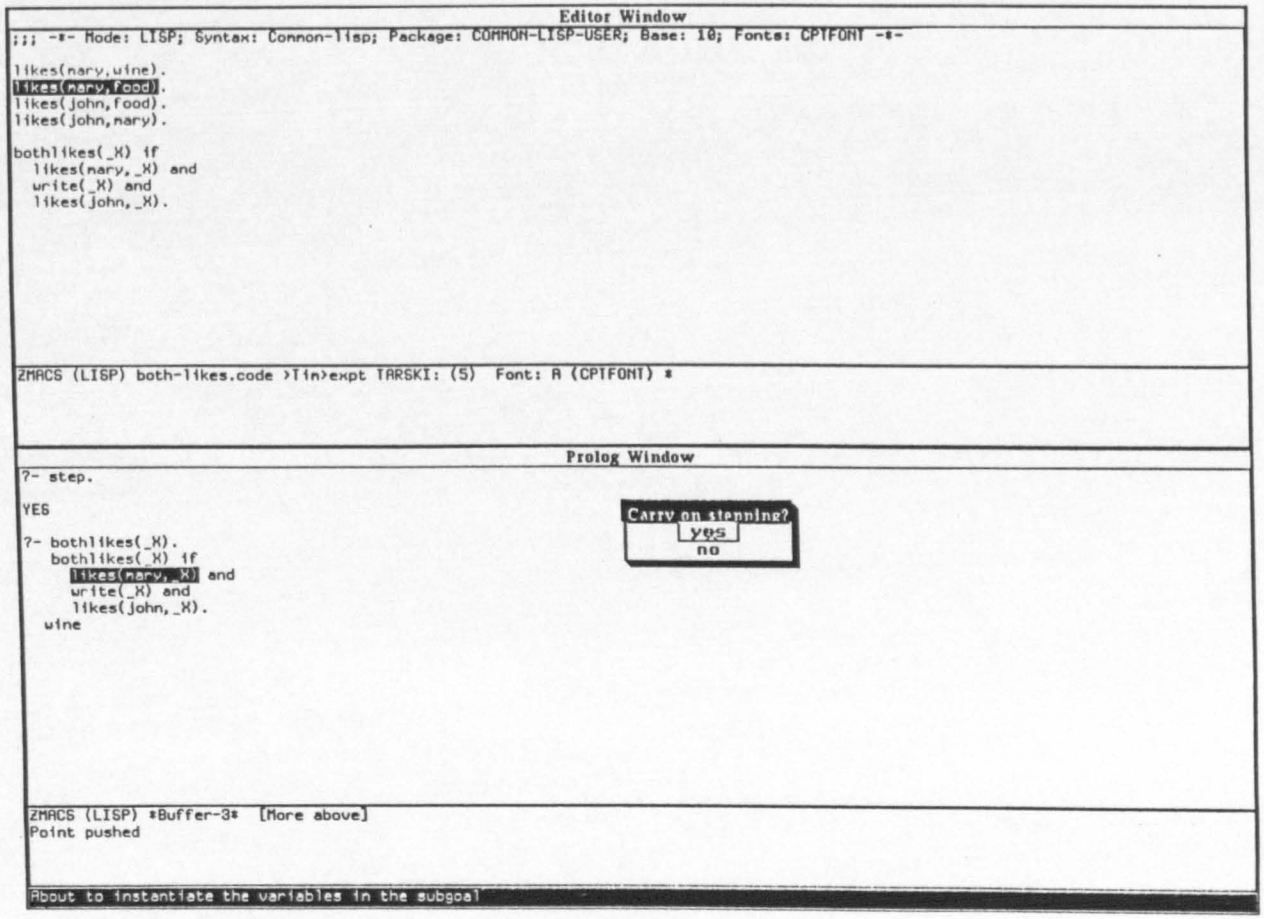

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(nary,wine).
likes(nary,food).
likes(john,food).
likes(john,nary).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(nary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- about to uninstantiate variables which were bound here last time round

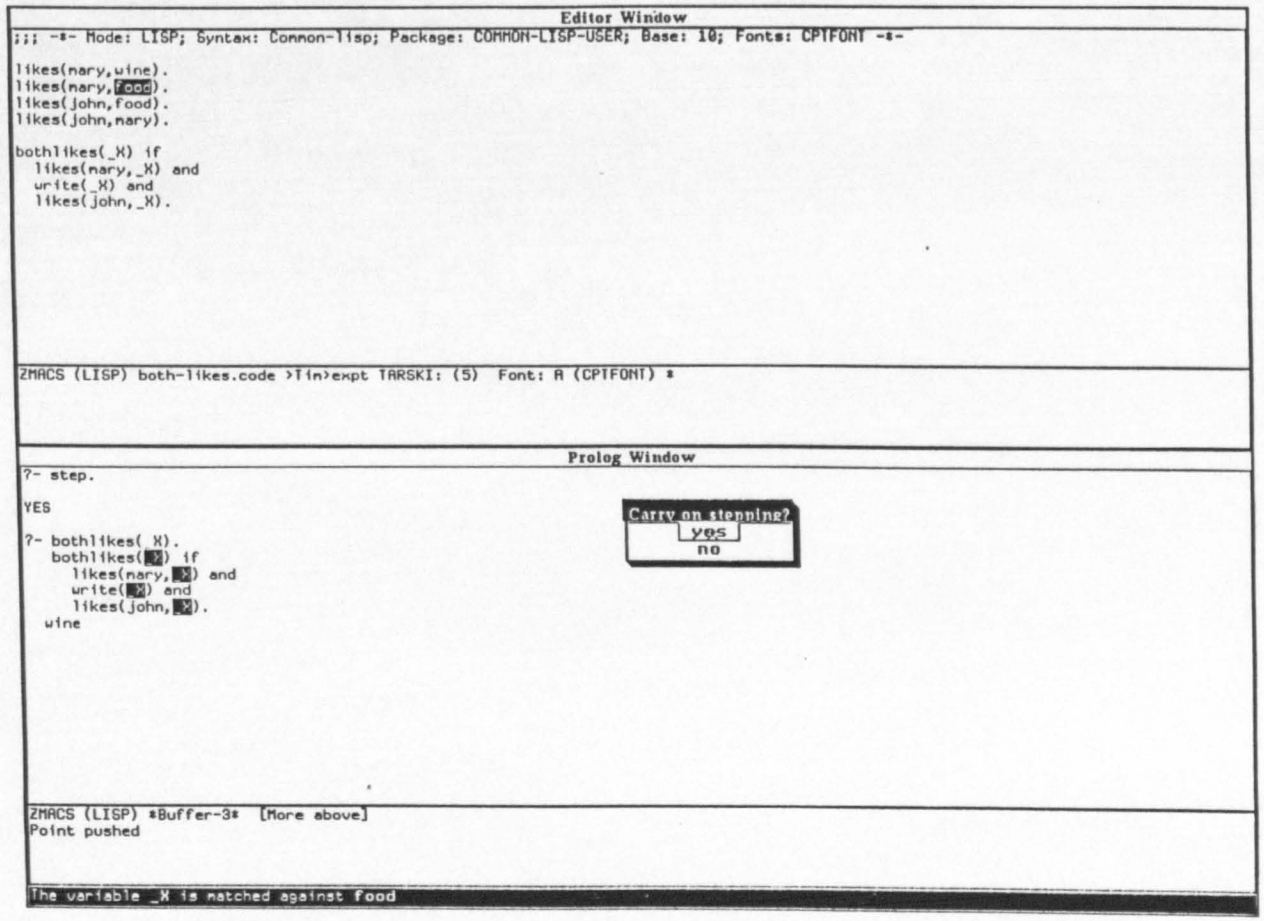

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(nary,wine).
likes(nary,food).
likes(john,food).
likes(john,nary).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(wine) if
likes(nary,wine) and
write(wine) and
likes(john,wine).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- about to uninstantiate _X which is instantiated to WINE

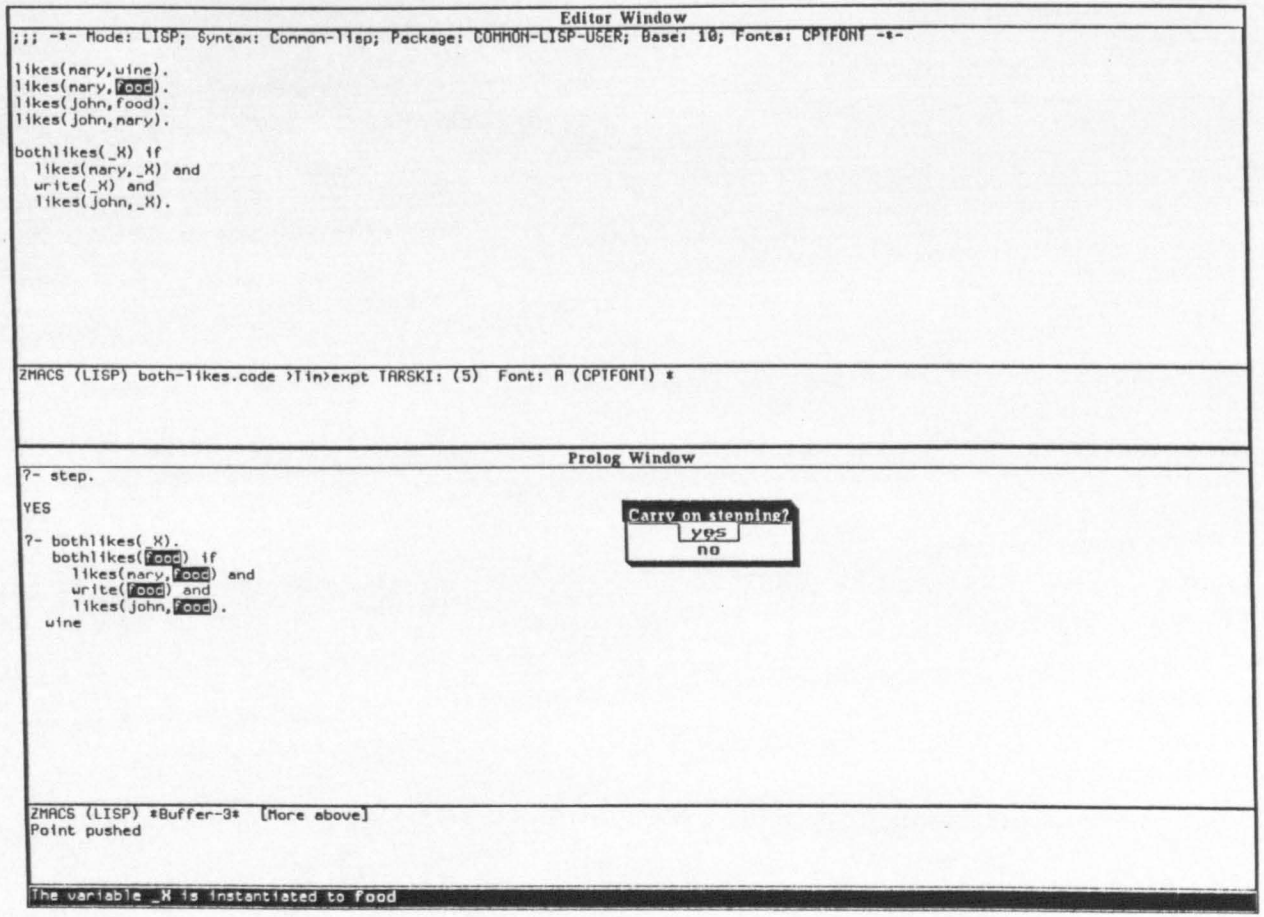

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- uninstantiating WINE back to _X

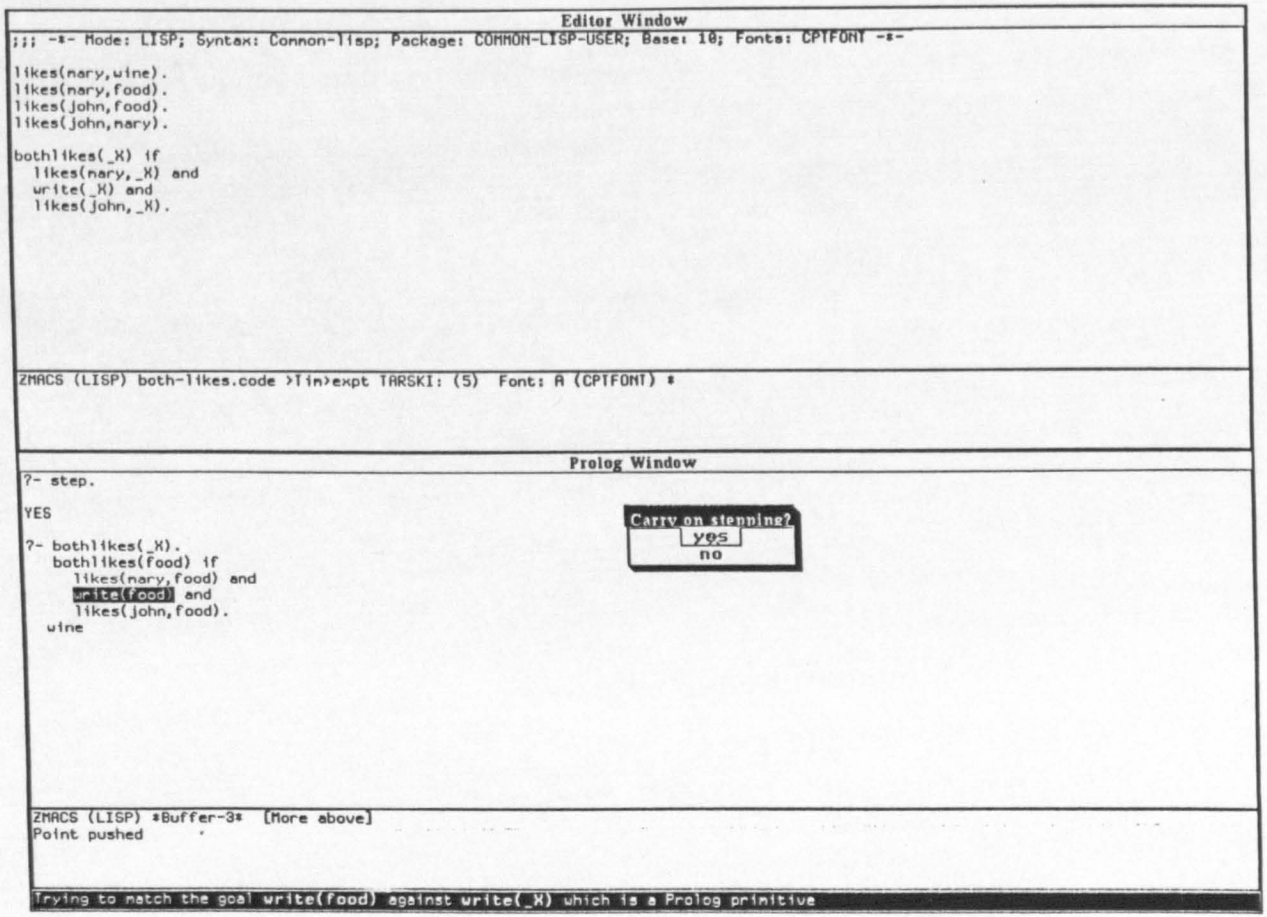

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(mary,_X) against likes(mary,food)

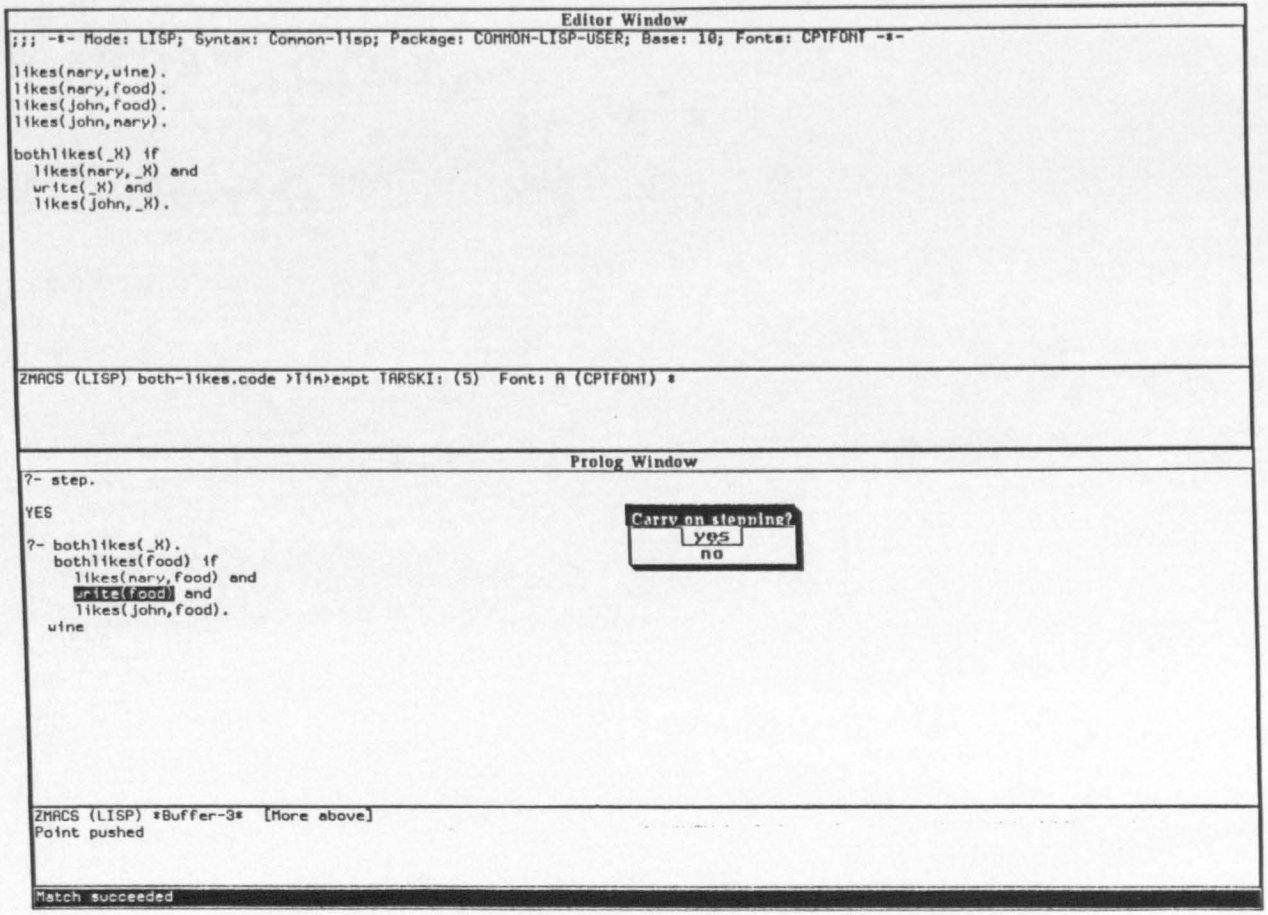

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >(in)expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded

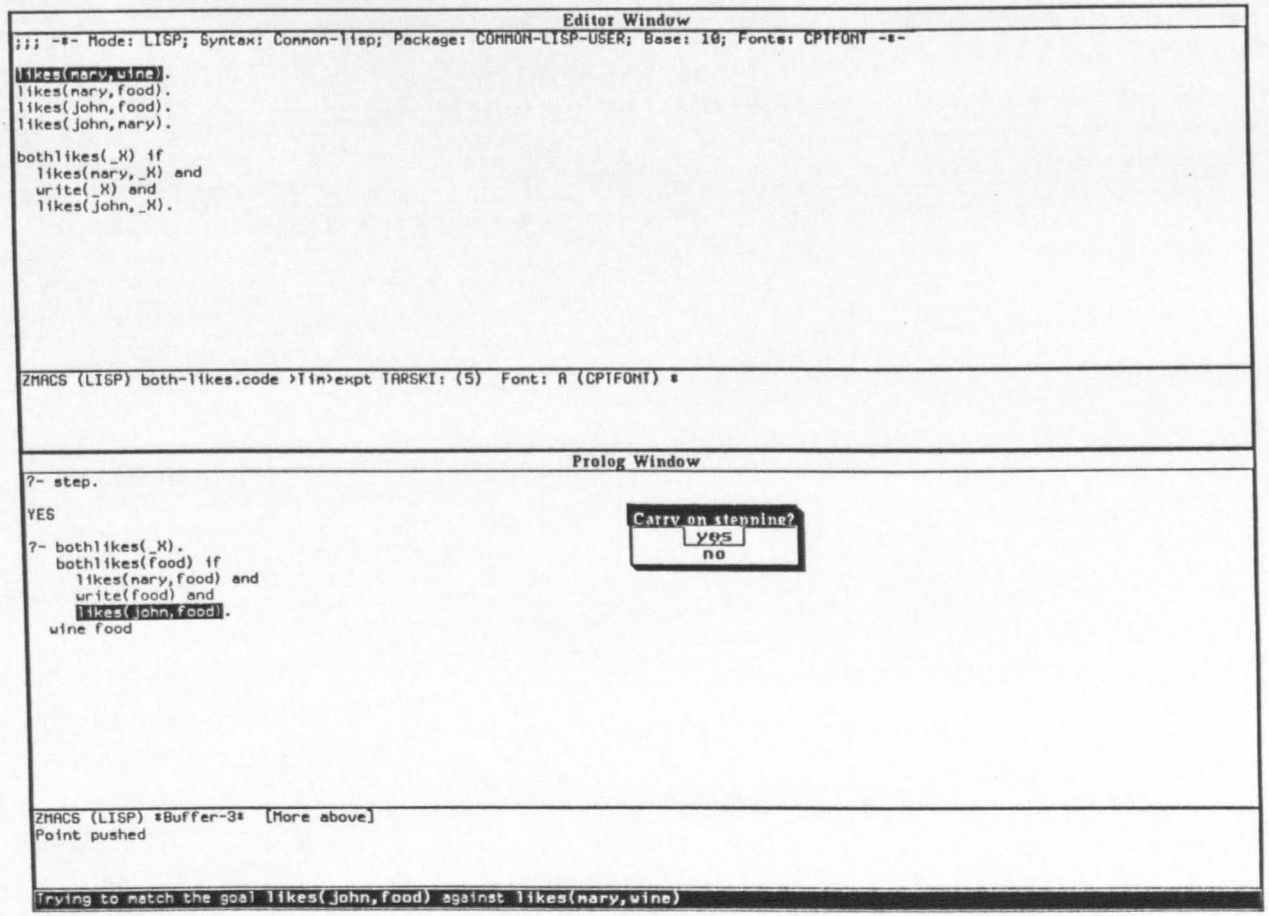

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
About to instantiate the variables in the subgoal

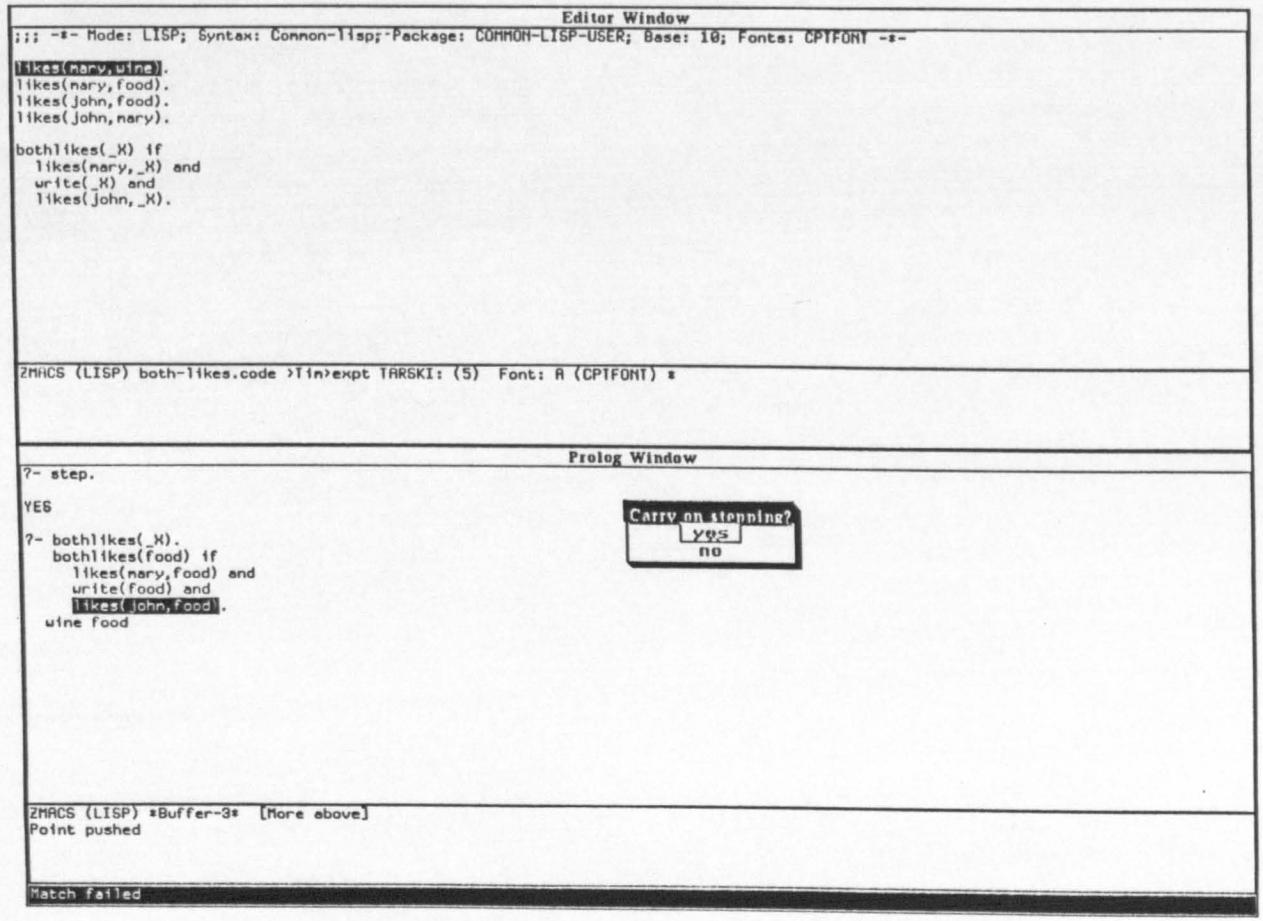

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(nary,wine).
likes(nary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes( X).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _X is matched against food

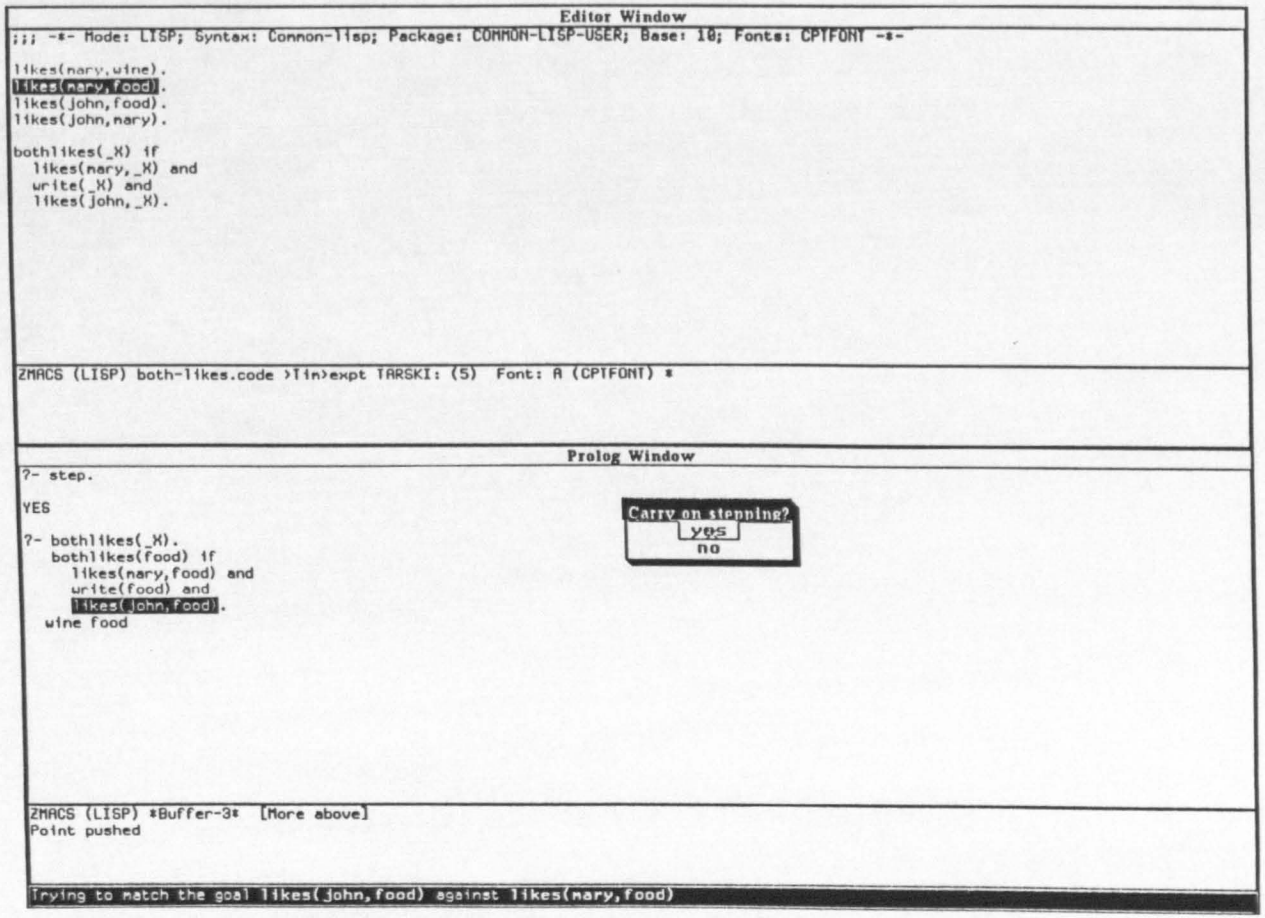

Editor Window
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _X is instantiated to food

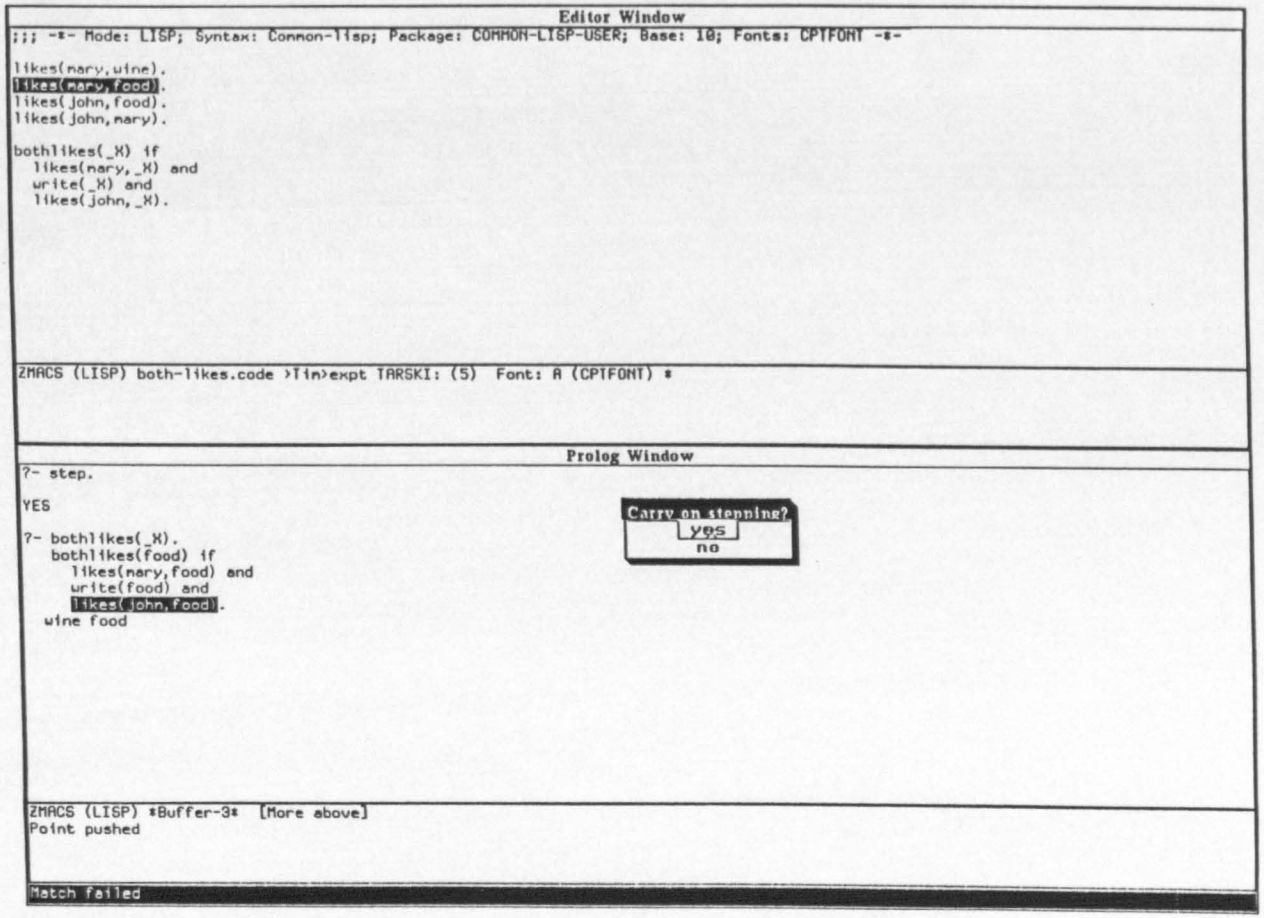

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal write(food) against write(_X) which is a Prolog primitive

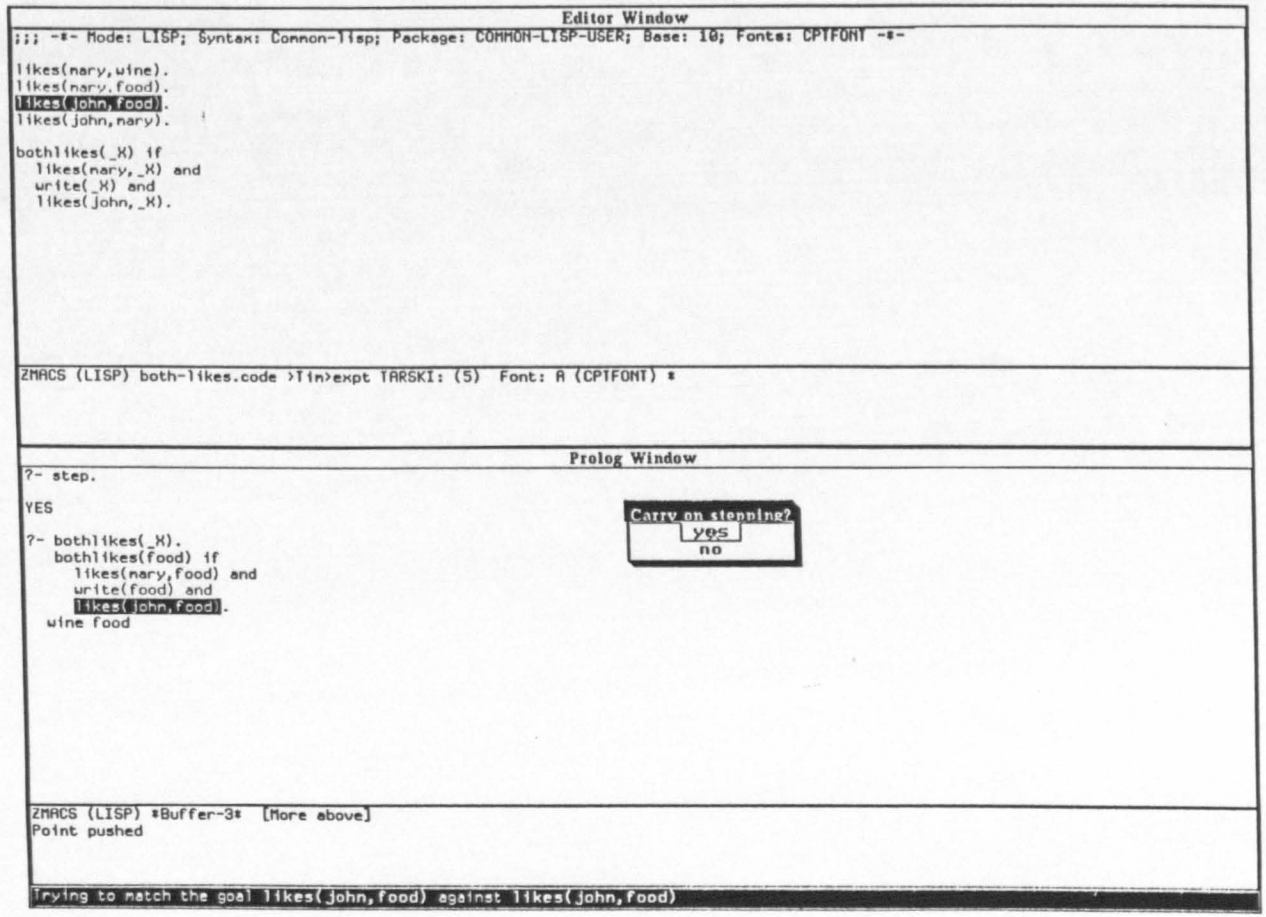

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(nary,wine).
likes(nary,food).
likes(john,food).
likes(john,nary).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(nary,food) and
write(food) and
likes(john,food).
wine
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded

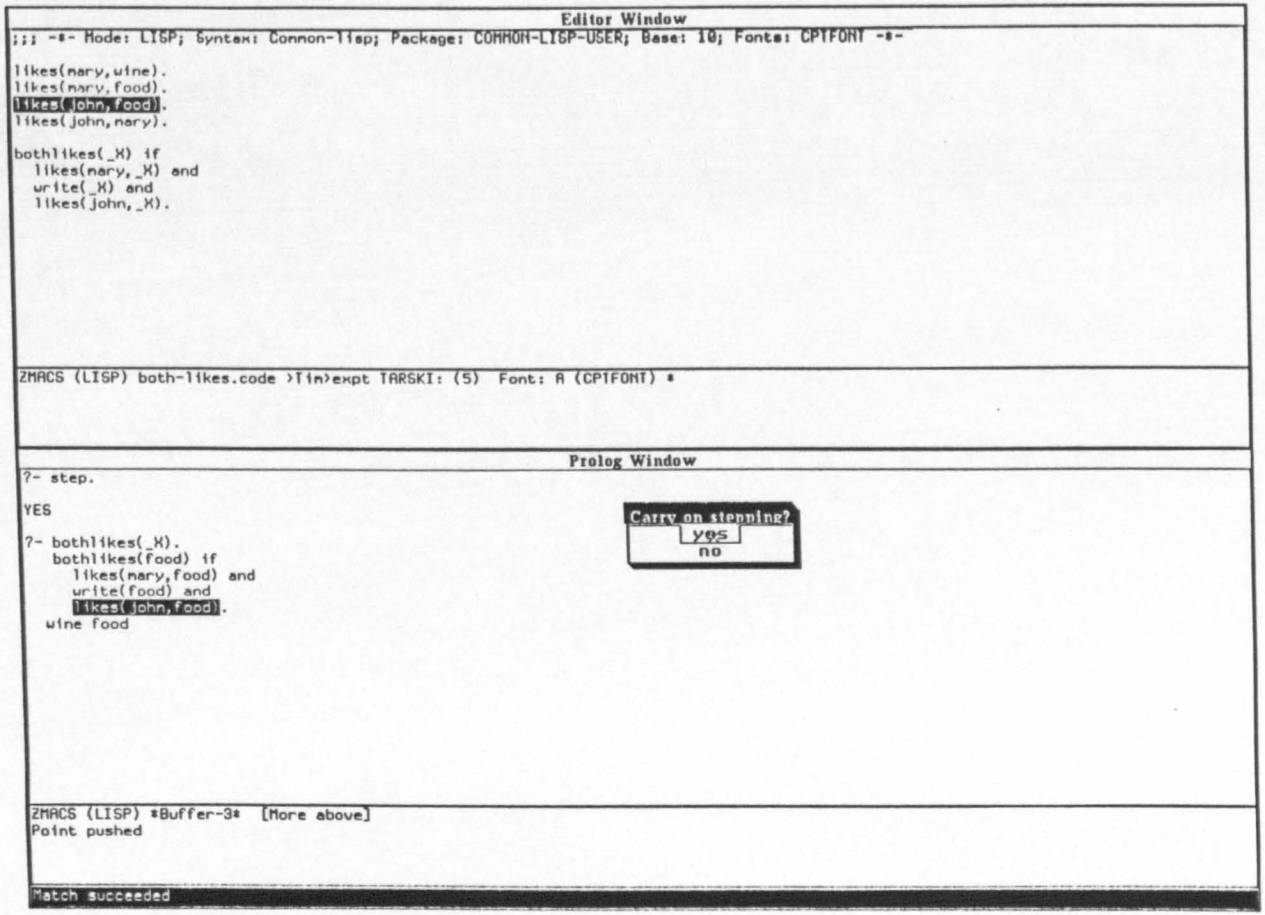

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine food
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(john,food) against likes(mary,wine)

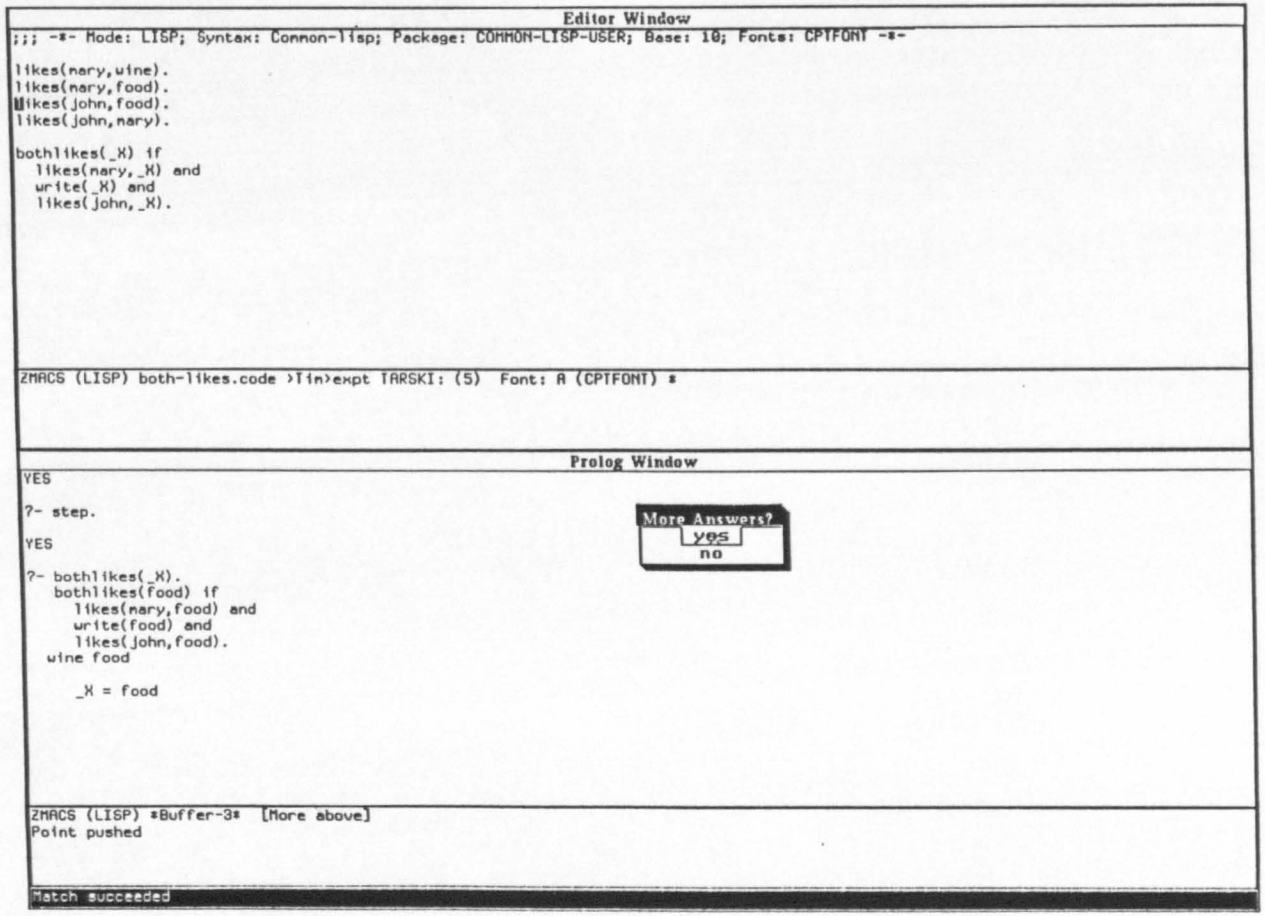

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine food
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed

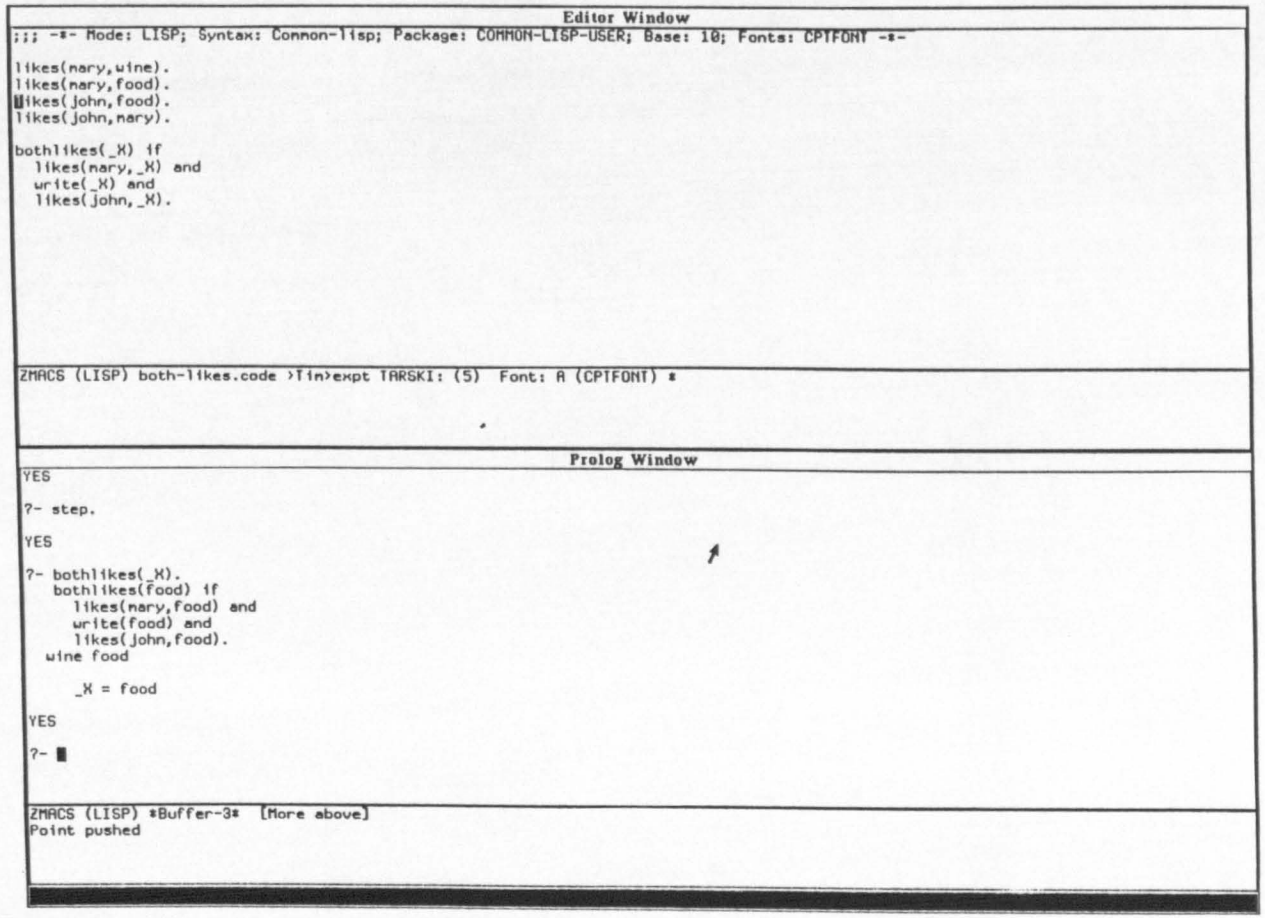

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine food
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(john,food) against likes(mary,food)


Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine food
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed


Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine food
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal likes(john,food) against likes(john,food)


Editor Window
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tim>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine food
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded

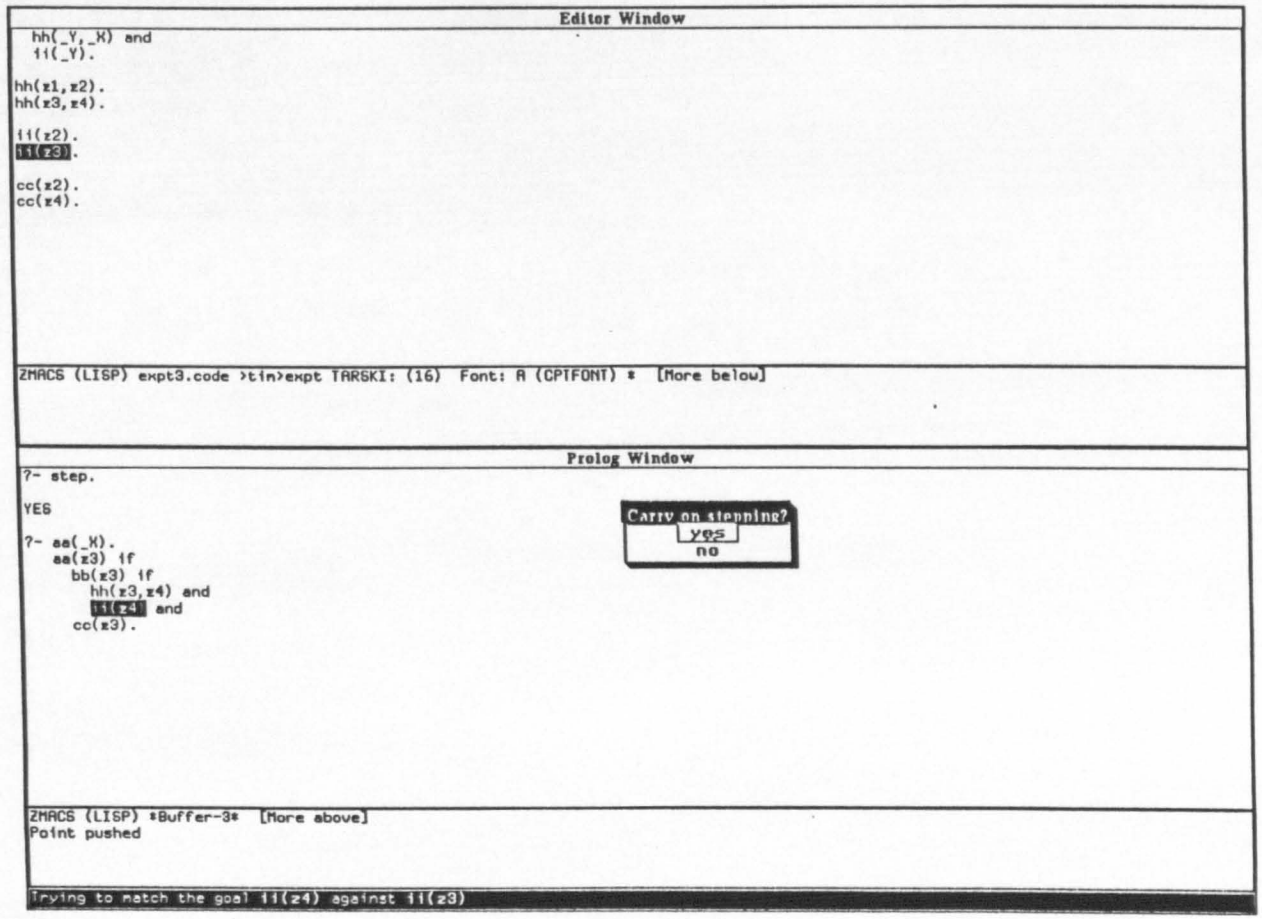

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(nary,wine).
likes(nary,food).
likes(john,food).
likes(john,nary).
bothlikes(_X) if
likes(nary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
YES
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(nary,food) and
write(food) and
likes(john,food).
wine food
_X = food
More Answers?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded

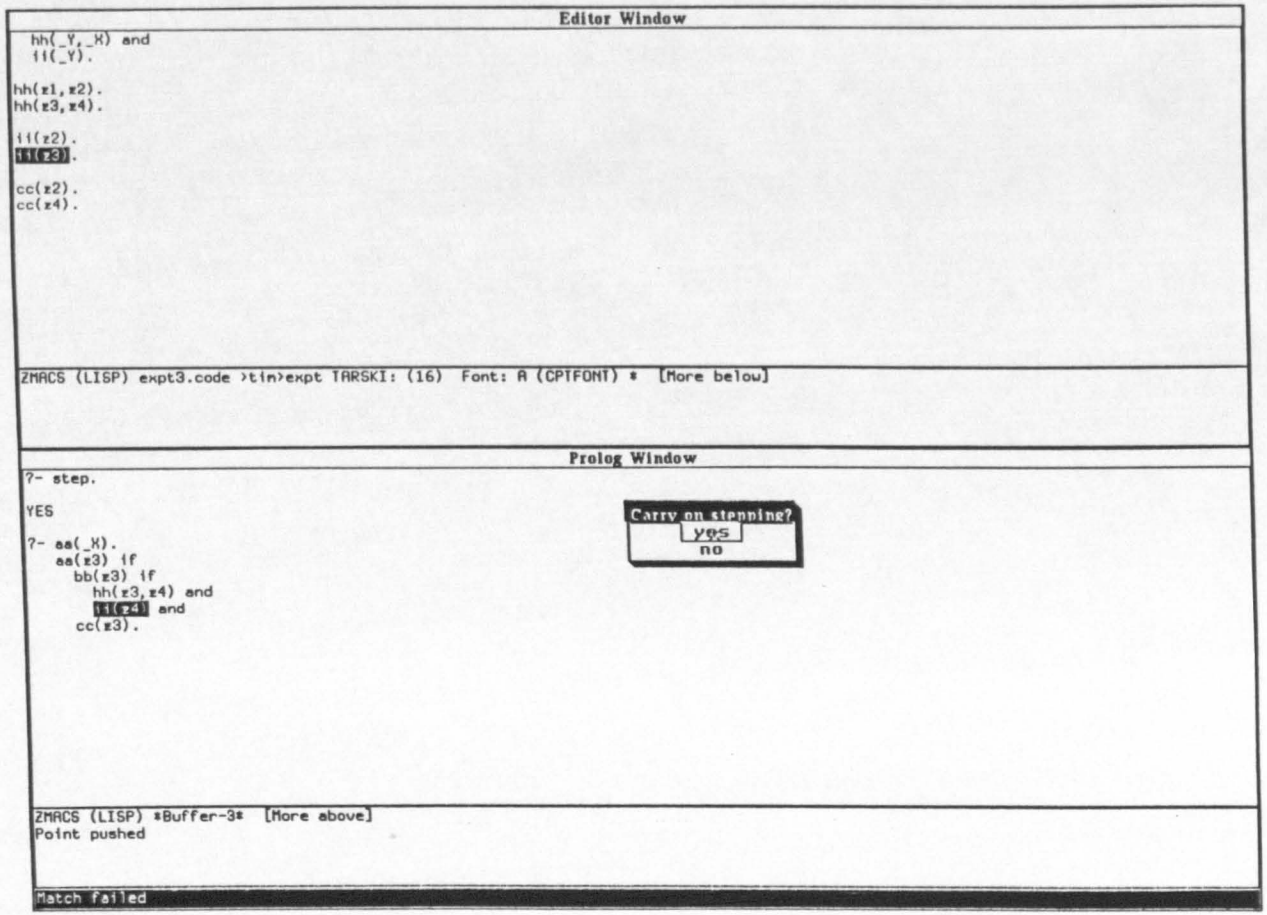

Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
likes(mary,wine).
likes(mary,food).
likes(john,food).
likes(john,mary).
bothlikes(_X) if
likes(mary,_X) and
write(_X) and
likes(john,_X).
ZMACS (LISP) both-likes.code >Tin>expt TARSKI: (5) Font: A (CPTFONT) *
Prolog Window
YES
?- step.
YES
?- bothlikes(_X).
bothlikes(food) if
likes(mary,food) and
write(food) and
likes(john,food).
wine food
_X = food
YES
?-
ZMACS (LISP) *Buffer-3* [More above]
Point pushed

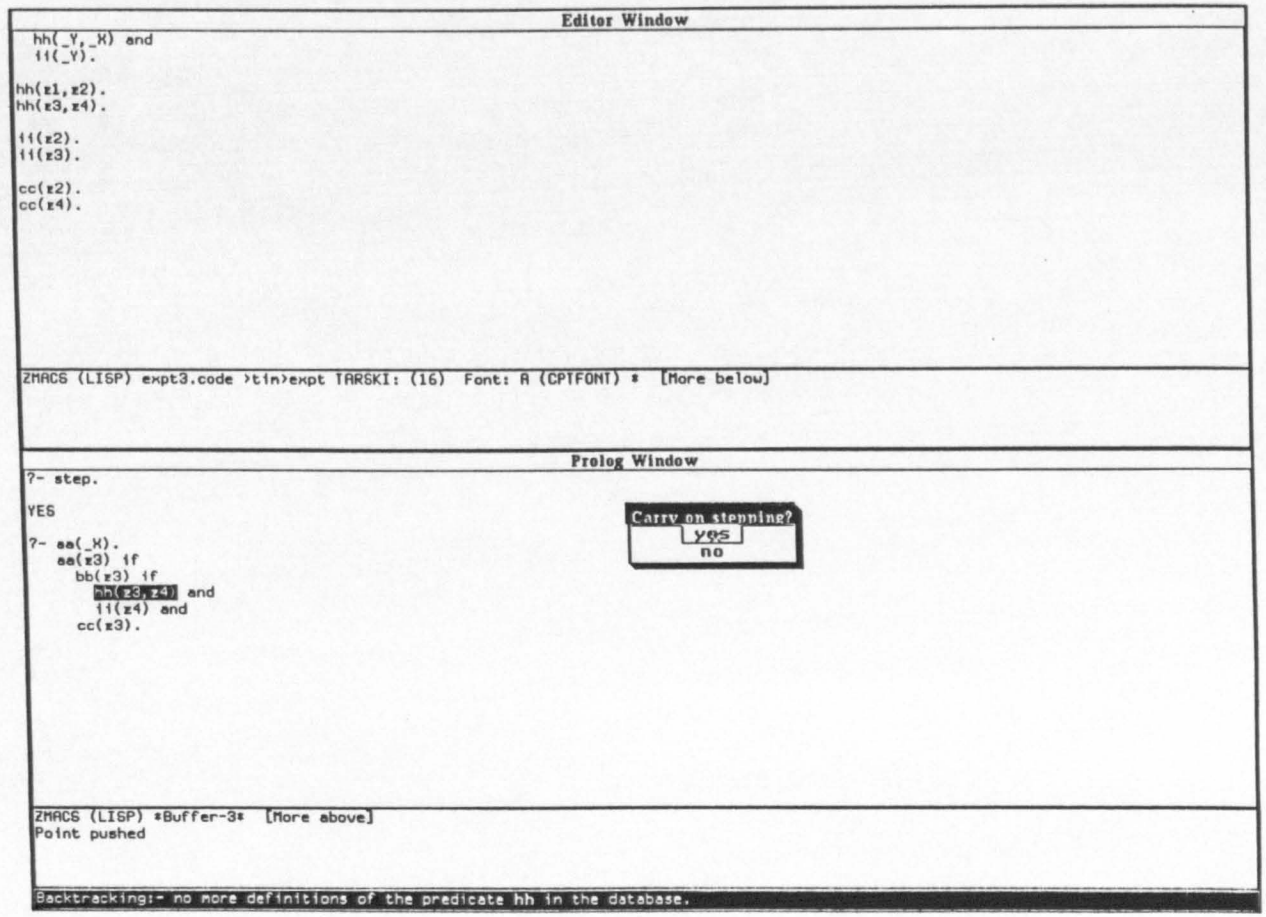
## APPENDIX-H

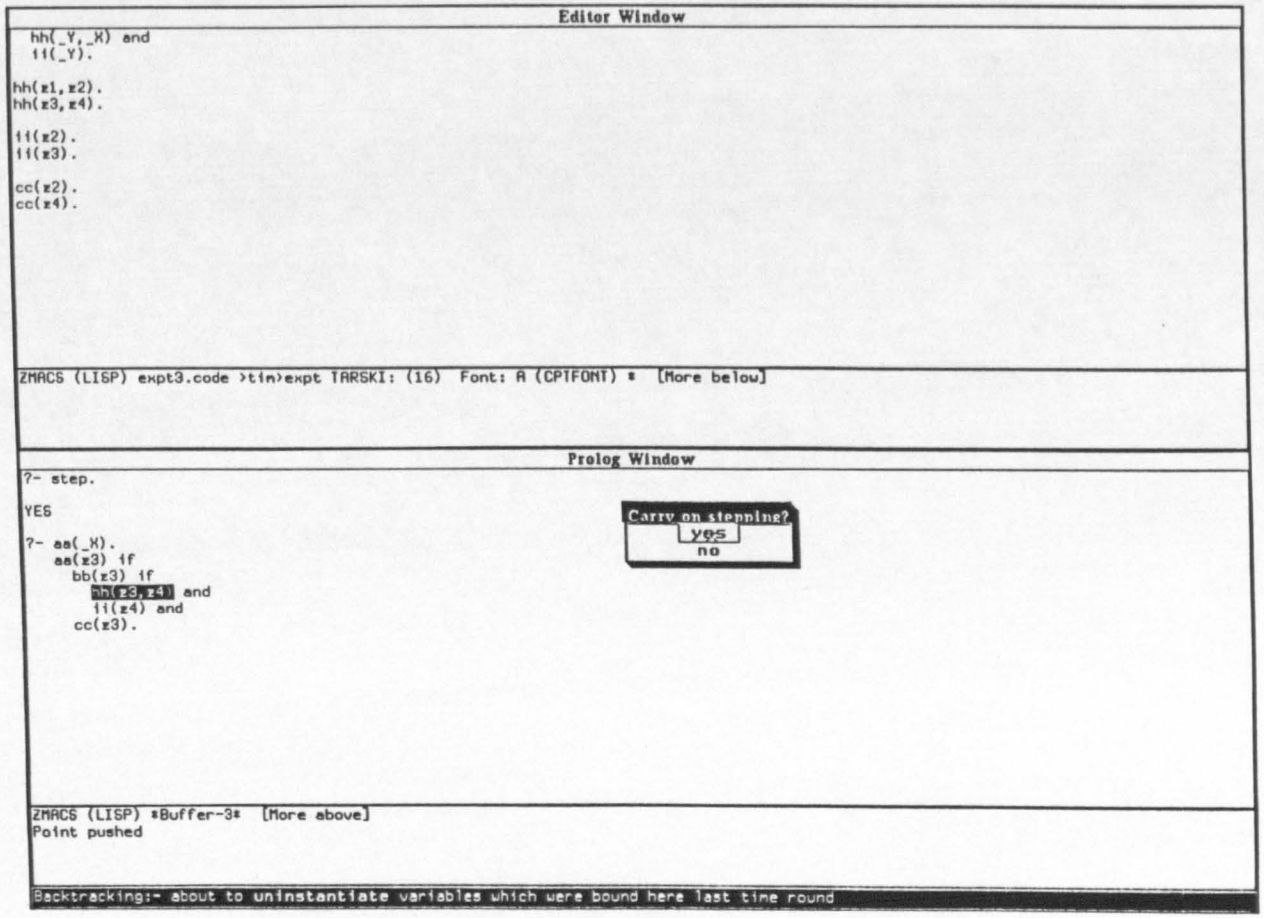
Appendix H presents an example of how APT displays backtracking in program execution. The example consists of an extract from the execution of the Prolog program listed below.

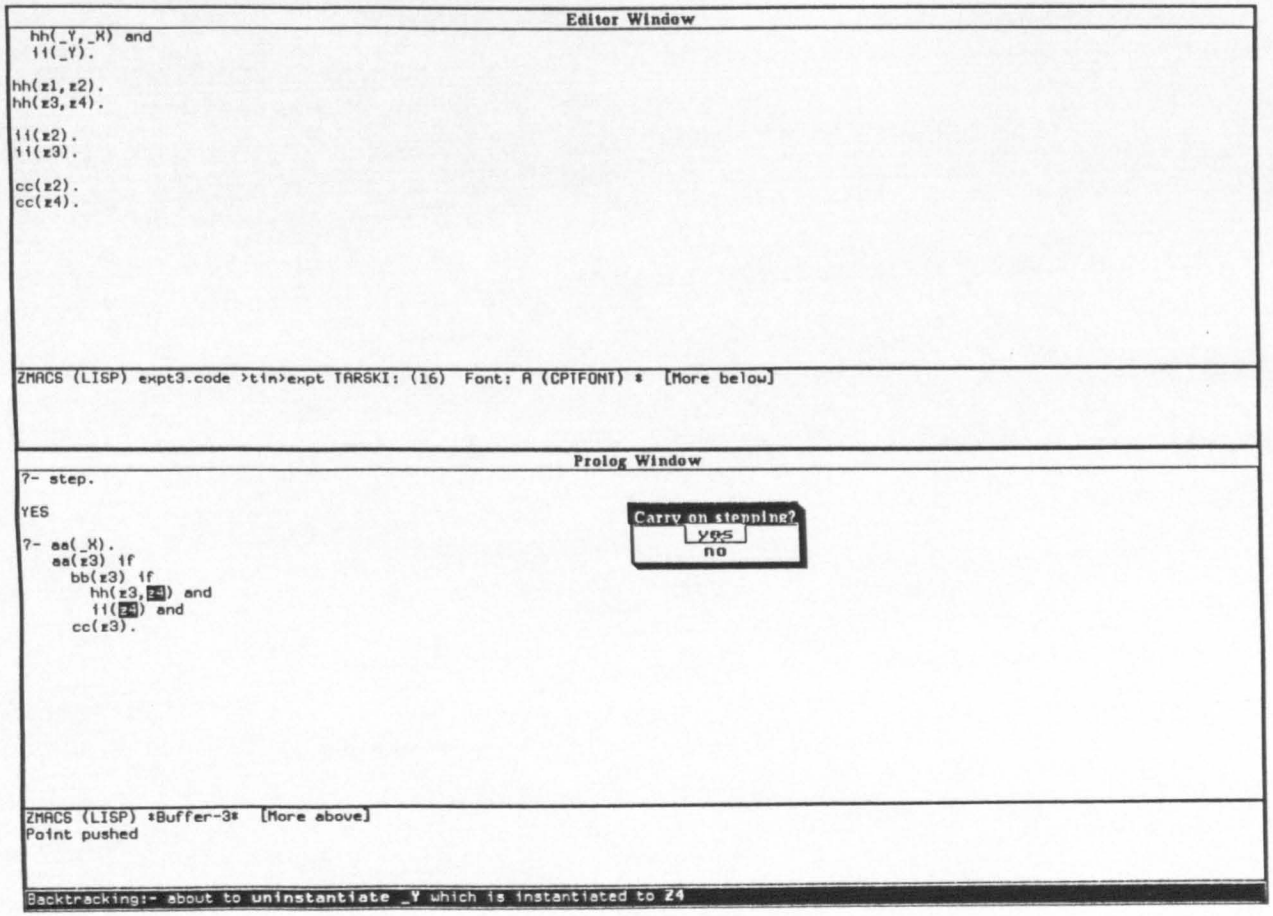
```
aa(_X) if
  bb(_X) and
  cc(_X).

bb(_X) if
  hh(_X,_Y) and
  ii(_Y).

bb(_X) if
  hh(_Y,_X) and
  ii(_Y).

hh(z1,z2).
hh(z3,z4).

ii(z2).
ii(z3).

cc(z2).
cc(z4).
```

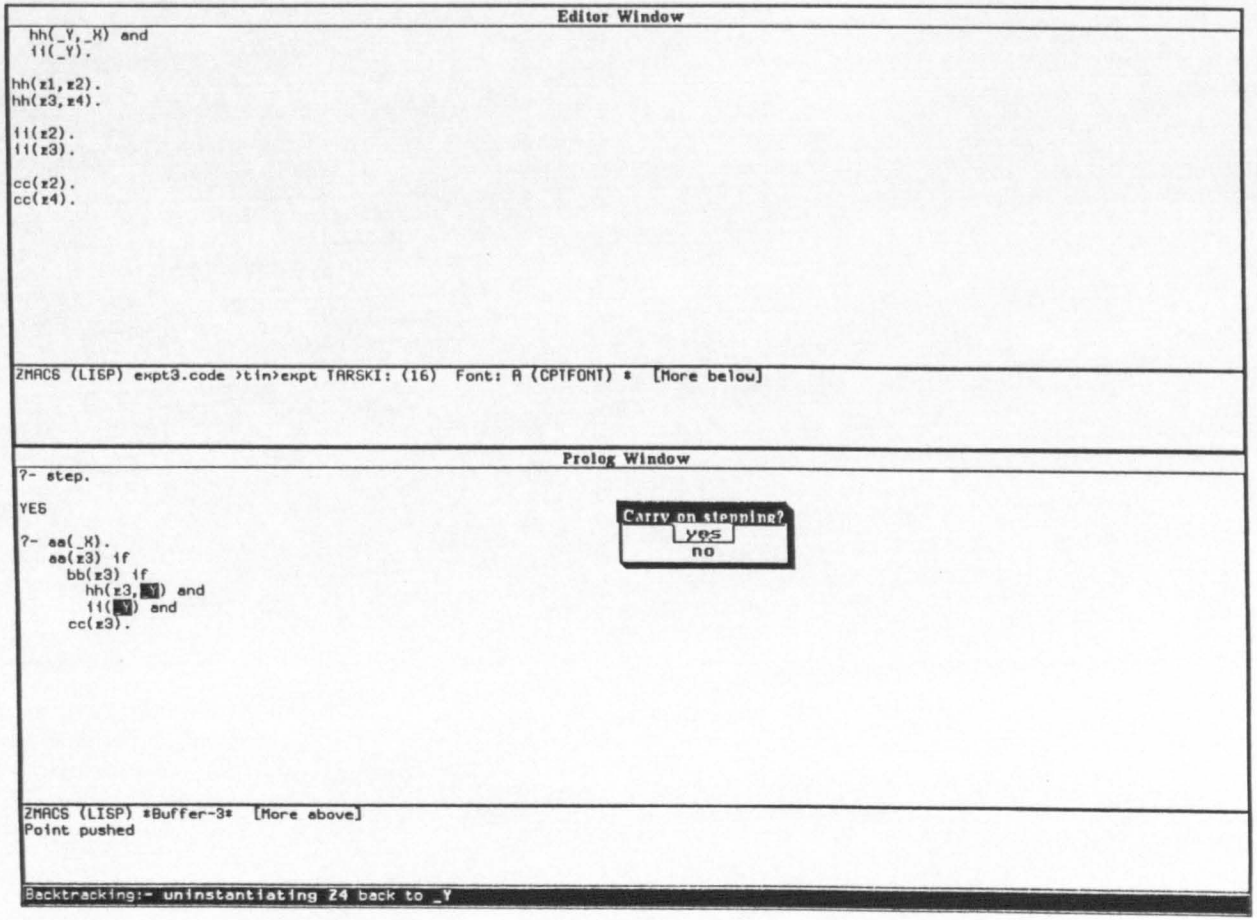

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(z3) if
bb(z3) if
hh(z3,z4) and
ii(z4) and
cc(z3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal ii(z4) against ii(z3)

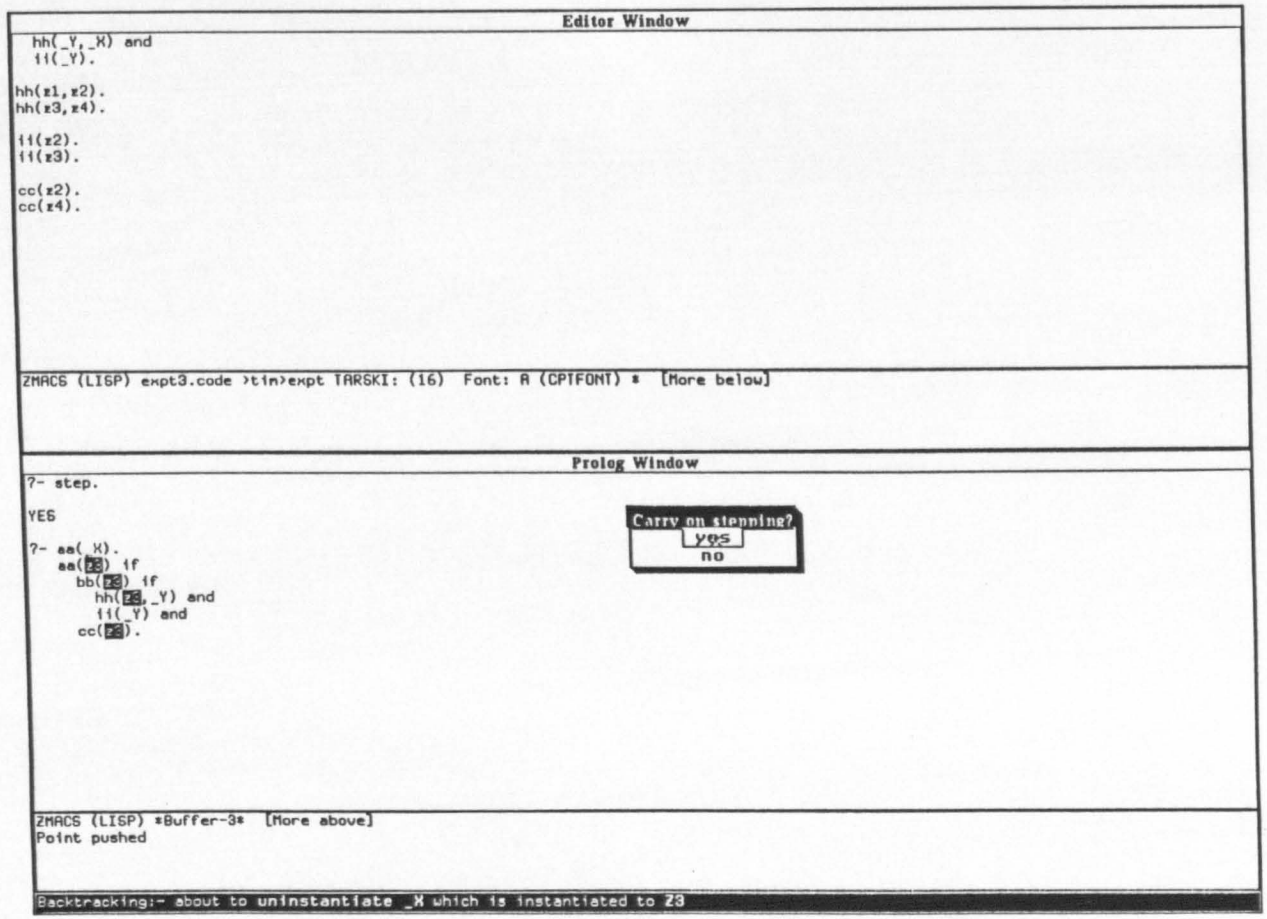

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(z3) if
bb(z3) if
hh(z3,z4) and
ii(z4) and
cc(z3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed

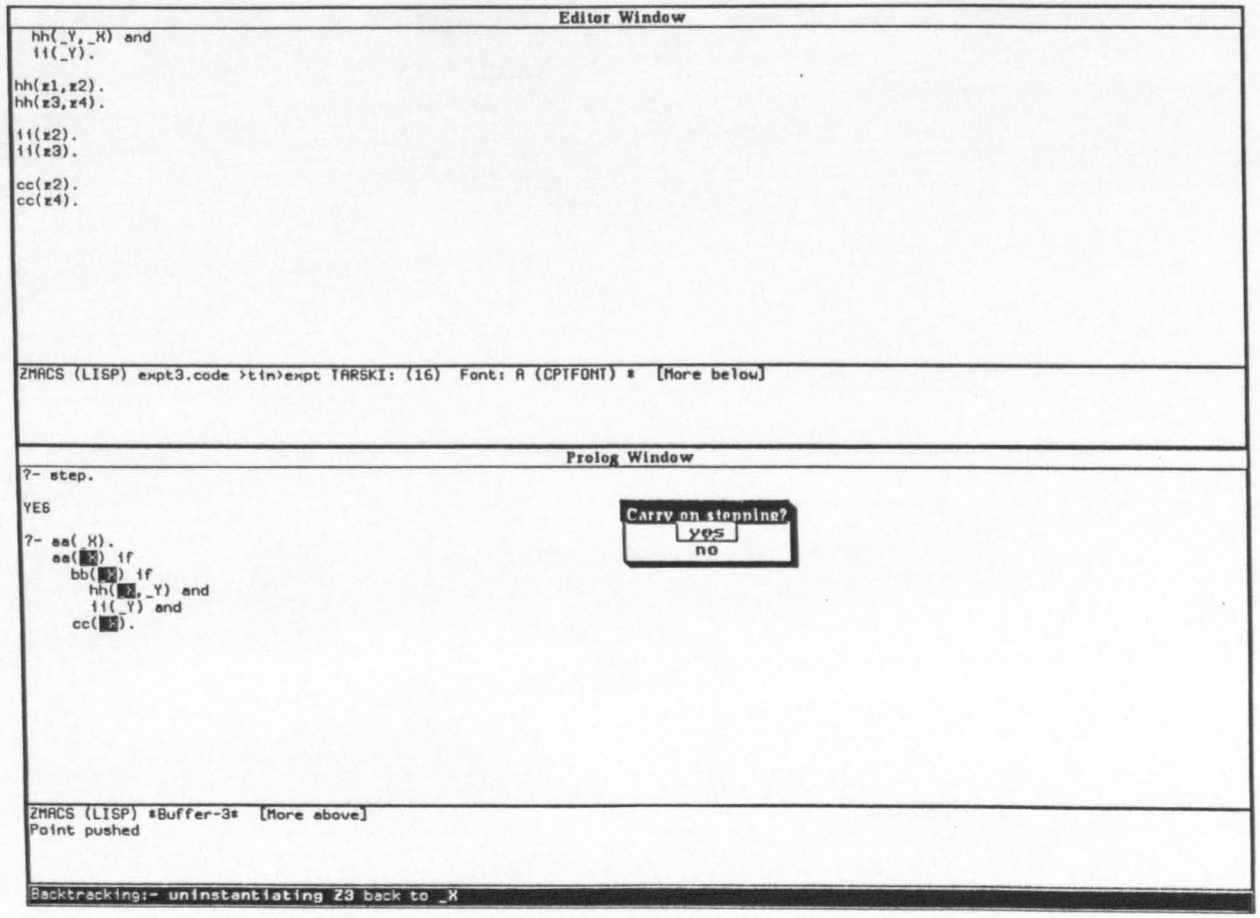

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tim>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(z3) if
bb(z3) if
hh(z3,z4) and
ii(z4) and
cc(z3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- no more definitions of the predicate hh in the database.

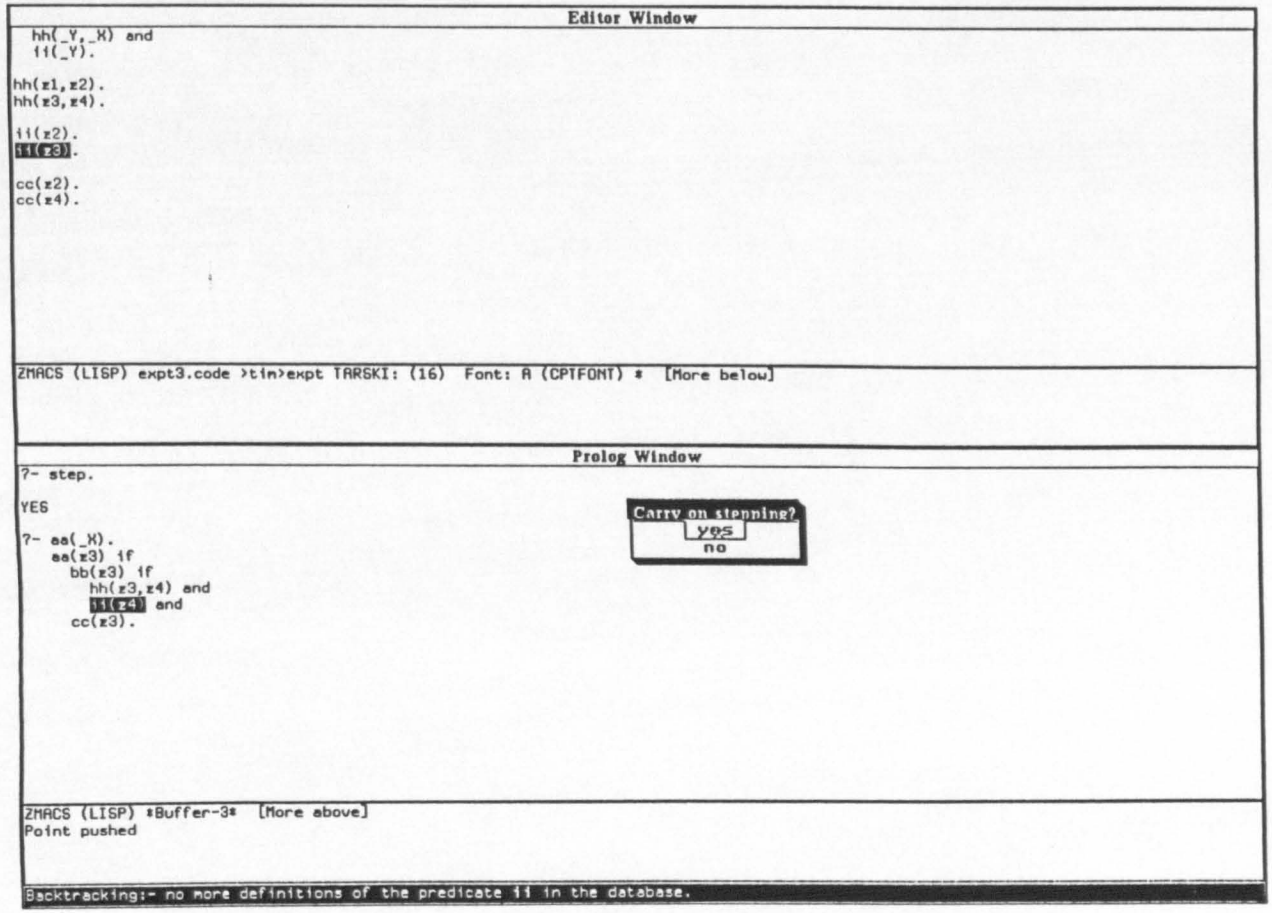

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(z3) if
bb(z3) if
hh(z3,z4) and
ii(z4) and
cc(z3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- about to uninstantiate variables which were bound here last time round

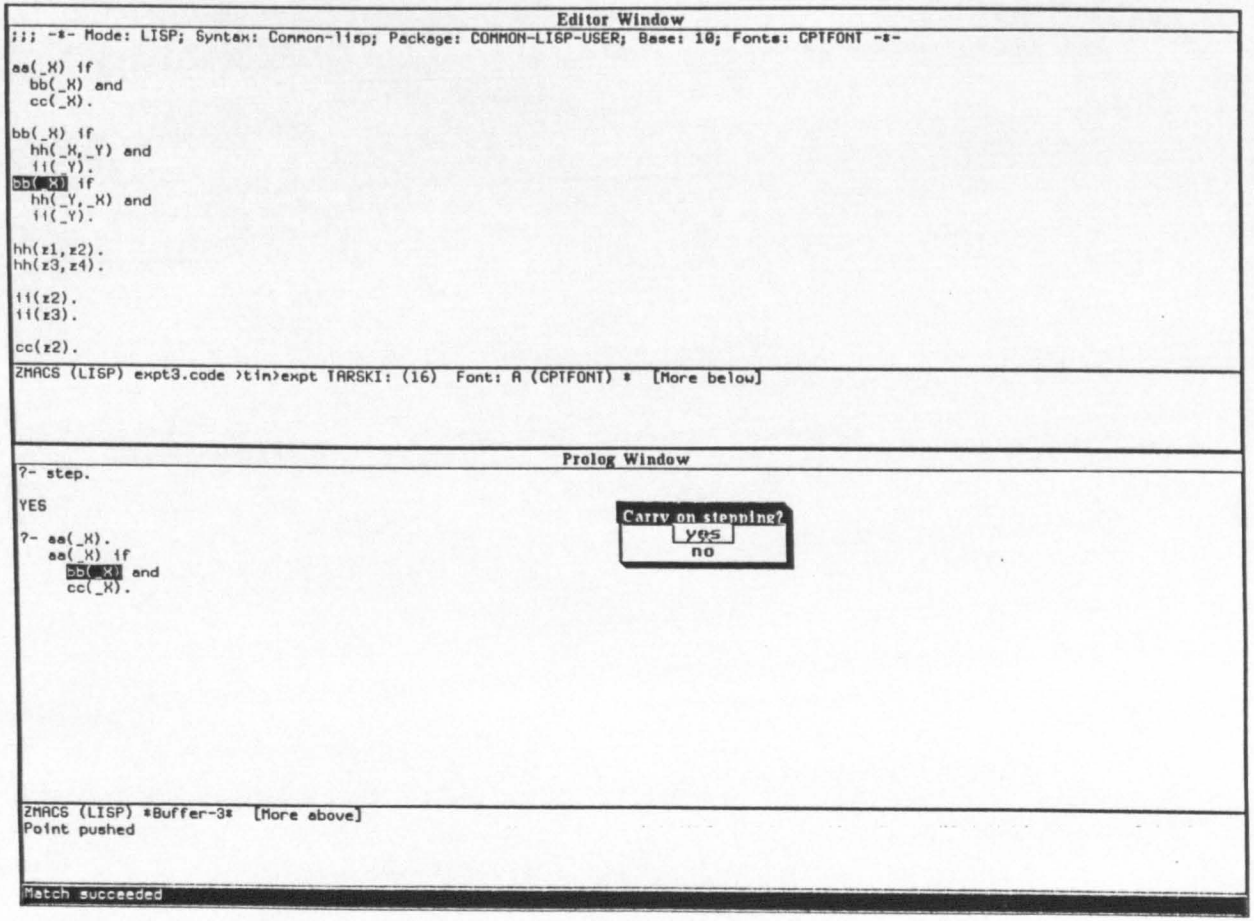

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(z3) if
bb(z3) if
hh(z3,z4) and
ii(z4) and
cc(z3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- about to uninstantiate _Y which is instantiated to z4

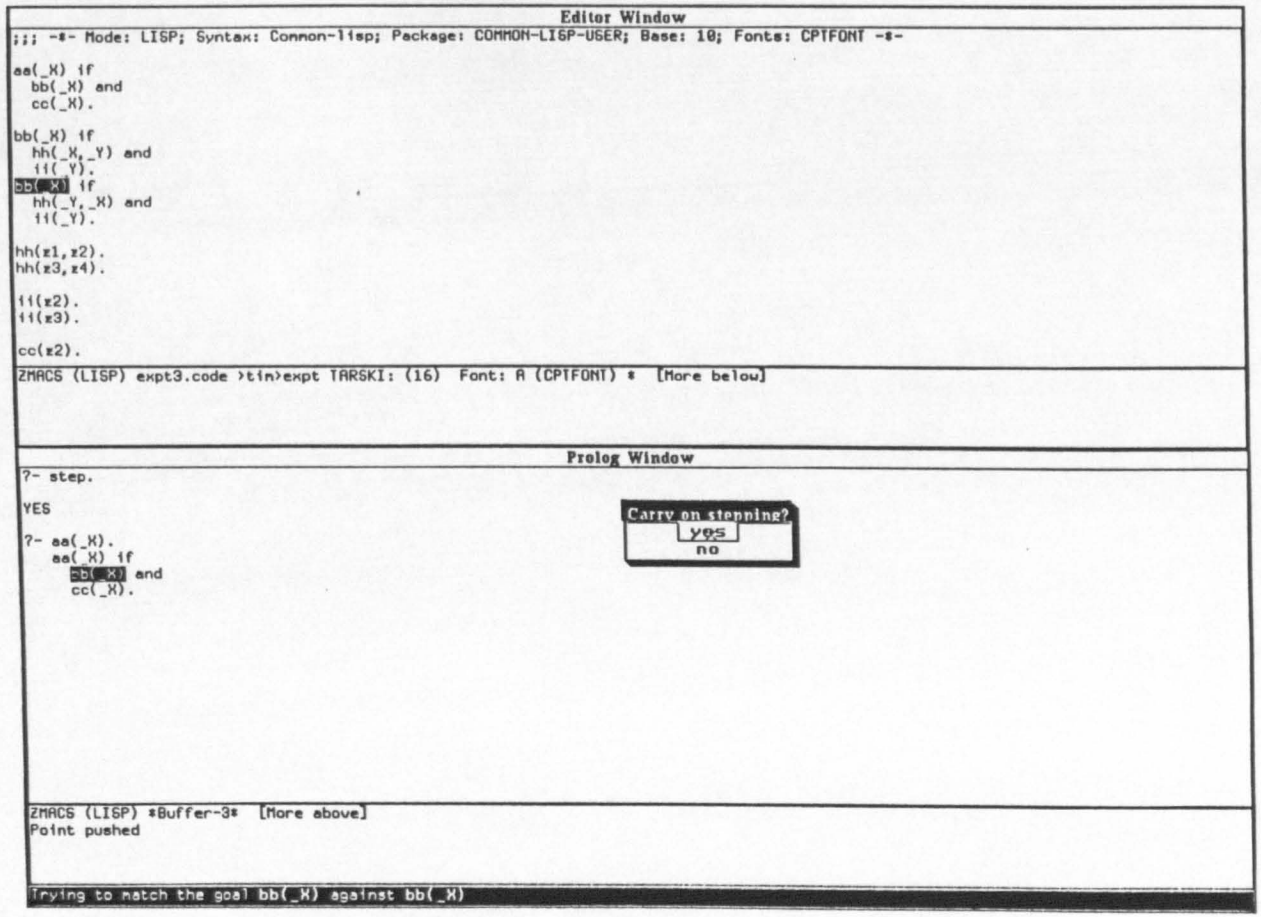

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(z3) if
bb(z3) if
hh(z3,_Y) and
ii(_Y) and
cc(z3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- uninstantiating Z4 back to _Y

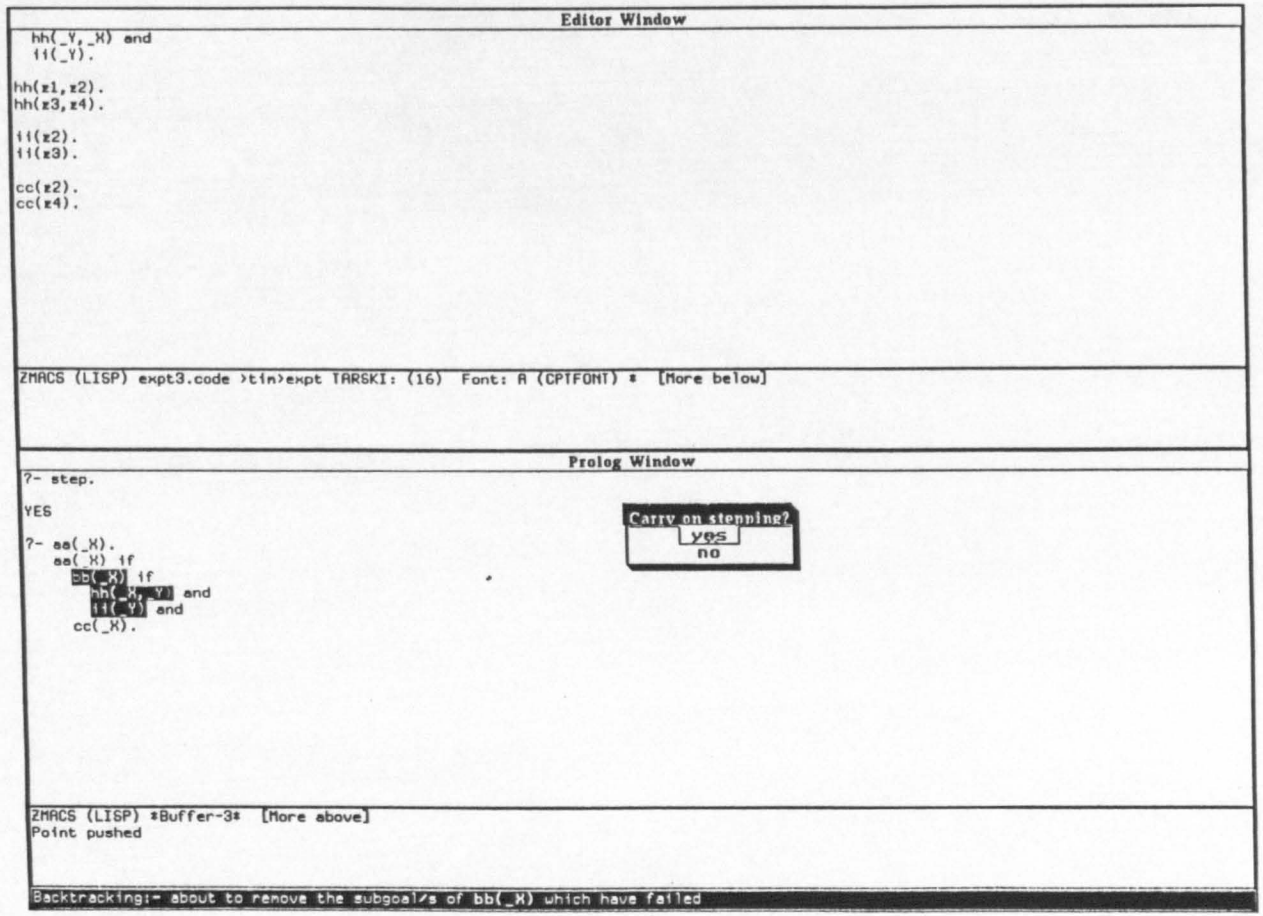

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(23) if
bb(23) if
hh(23,_Y) and
ii(_Y) and
cc(23).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- about to uninstantiate _X which is instantiated to 23

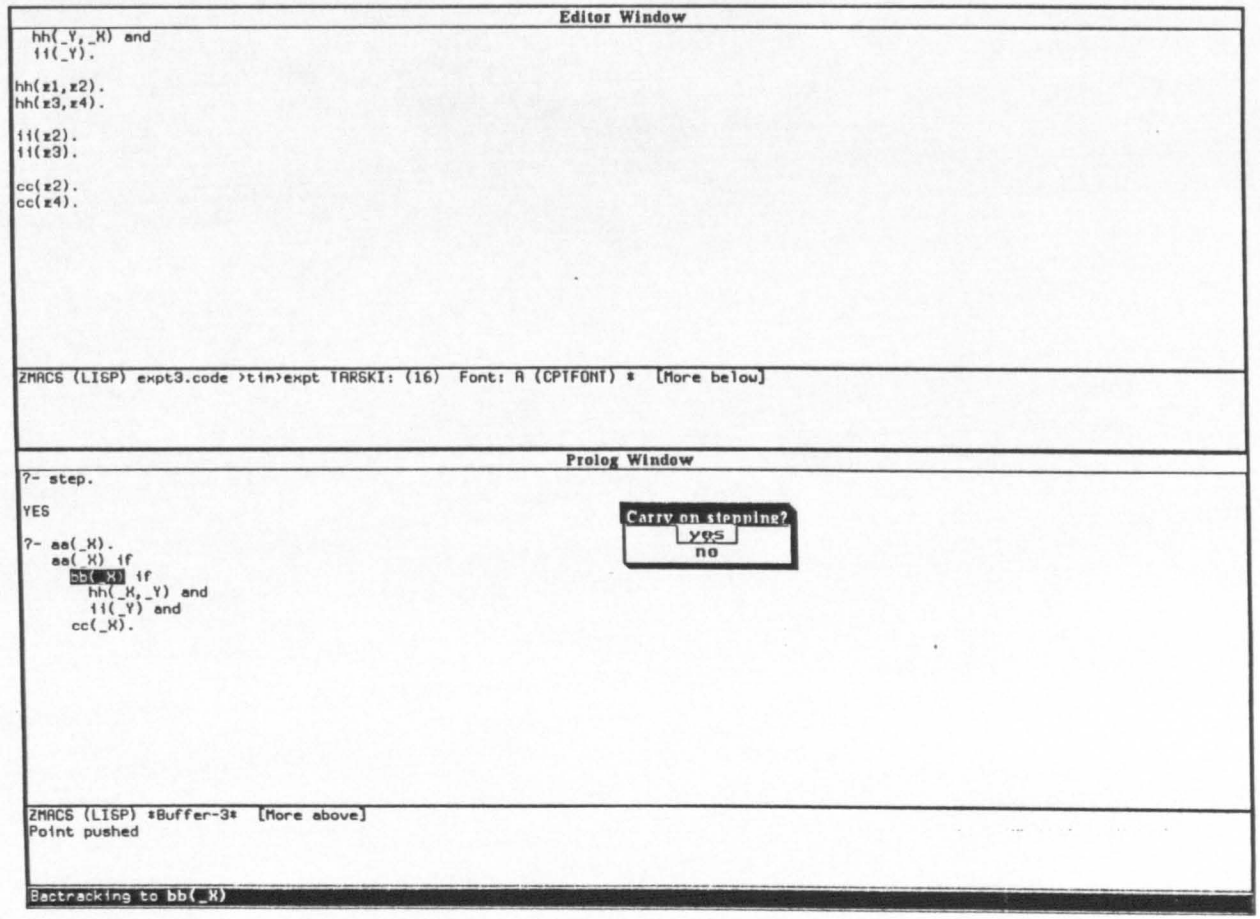

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(_X) if
bb(_X) if
hh(_X,_Y) and
ii(_Y) and
cc(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- uninstantiating Z3 back to _X


Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(z3) if
bb(z3) if
hh(z3,z4) and
ii(z4) and
cc(z3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- no more definitions of the predicate ii in the database.


Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
aa(_X) if
bb(_X) and
cc(_X).
bb(_X) if
hh(_X,_Y) and
ii(_Y).
bb(_X) if
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(_X) if
bb(_X) and
cc(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded


Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
aa(_X) if
bb(_X) and
cc(_X).
bb(_X) if
hh(_X,_Y) and
ii(_Y).
bb(_X) if
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(_X) if
bb(_X) and
cc(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal bb(_X) against bb(_X)

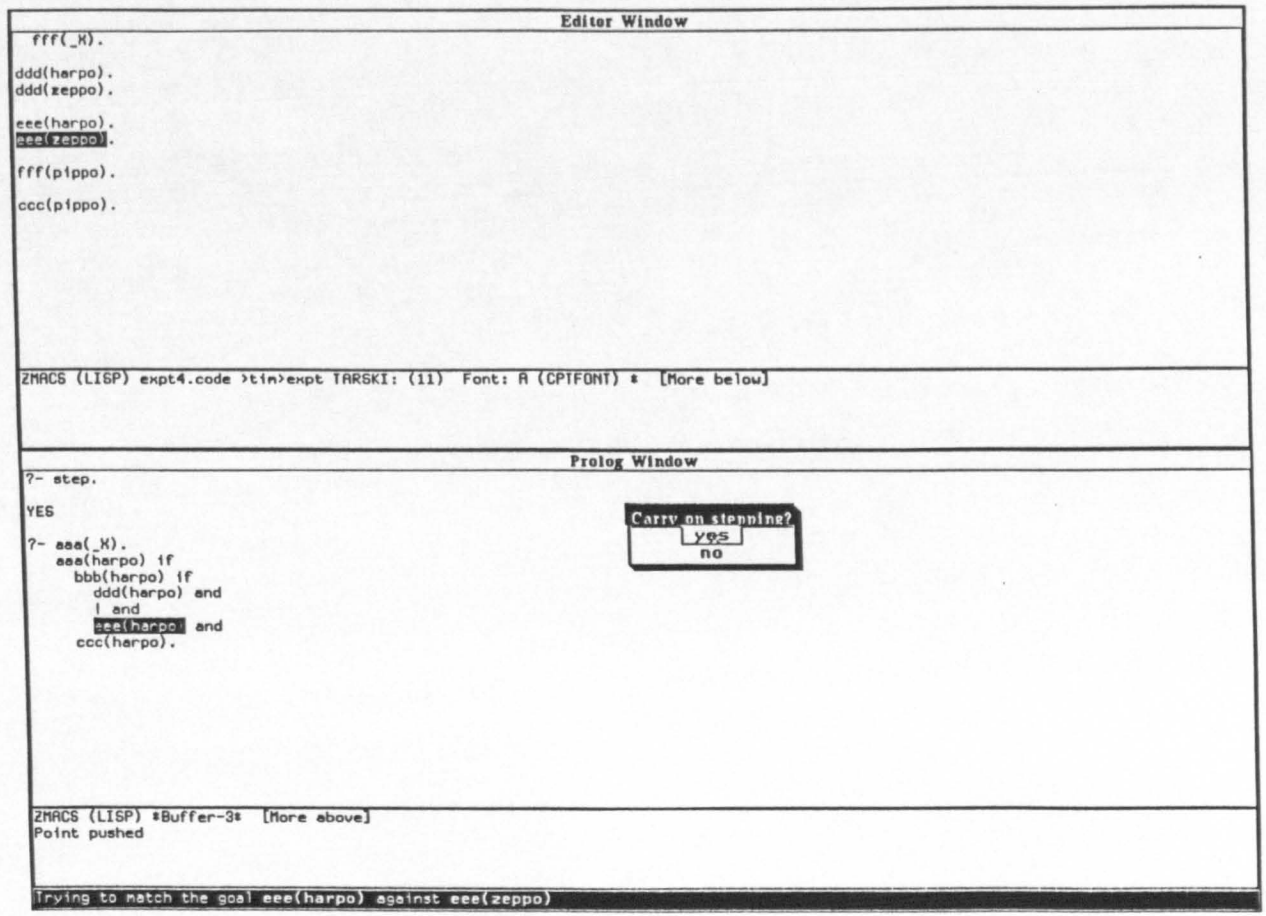

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tim>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(_X) if
bb(_X) if
hh(_X,_Y) and
ii(_Y) and
cc(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- about to remove the subgoal/s of bb(_X) which have failed

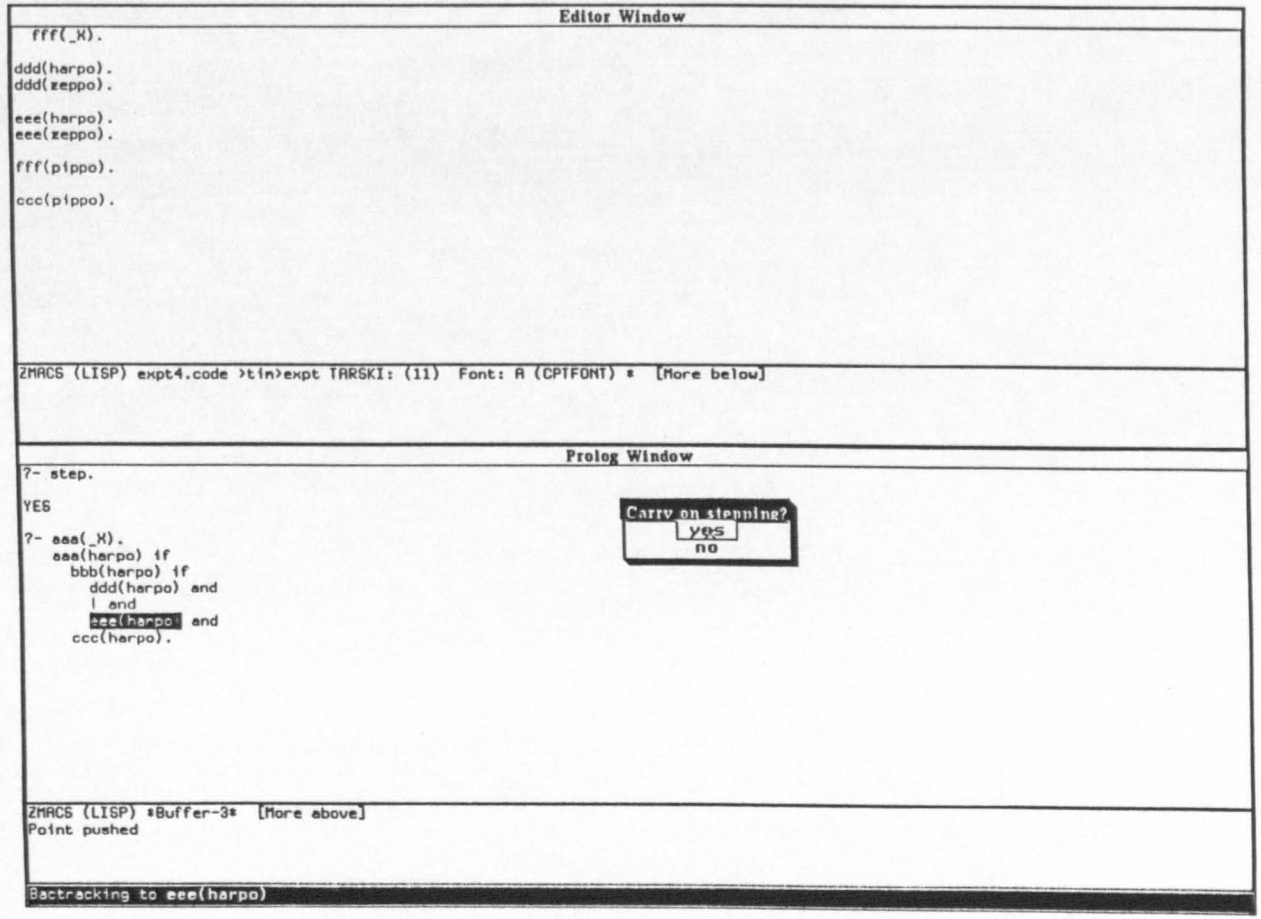

Editor Window
hh(_Y,_X) and
ii(_Y).
hh(z1,z2).
hh(z3,z4).
ii(z2).
ii(z3).
cc(z2).
cc(z4).
ZMACS (LISP) expt3.code >tin>expt TARSKI: (16) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aa(_X).
aa(_X) if
bb(_X) if
hh(_X,_Y) and
ii(_Y) and
cc(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking to bb(_X)

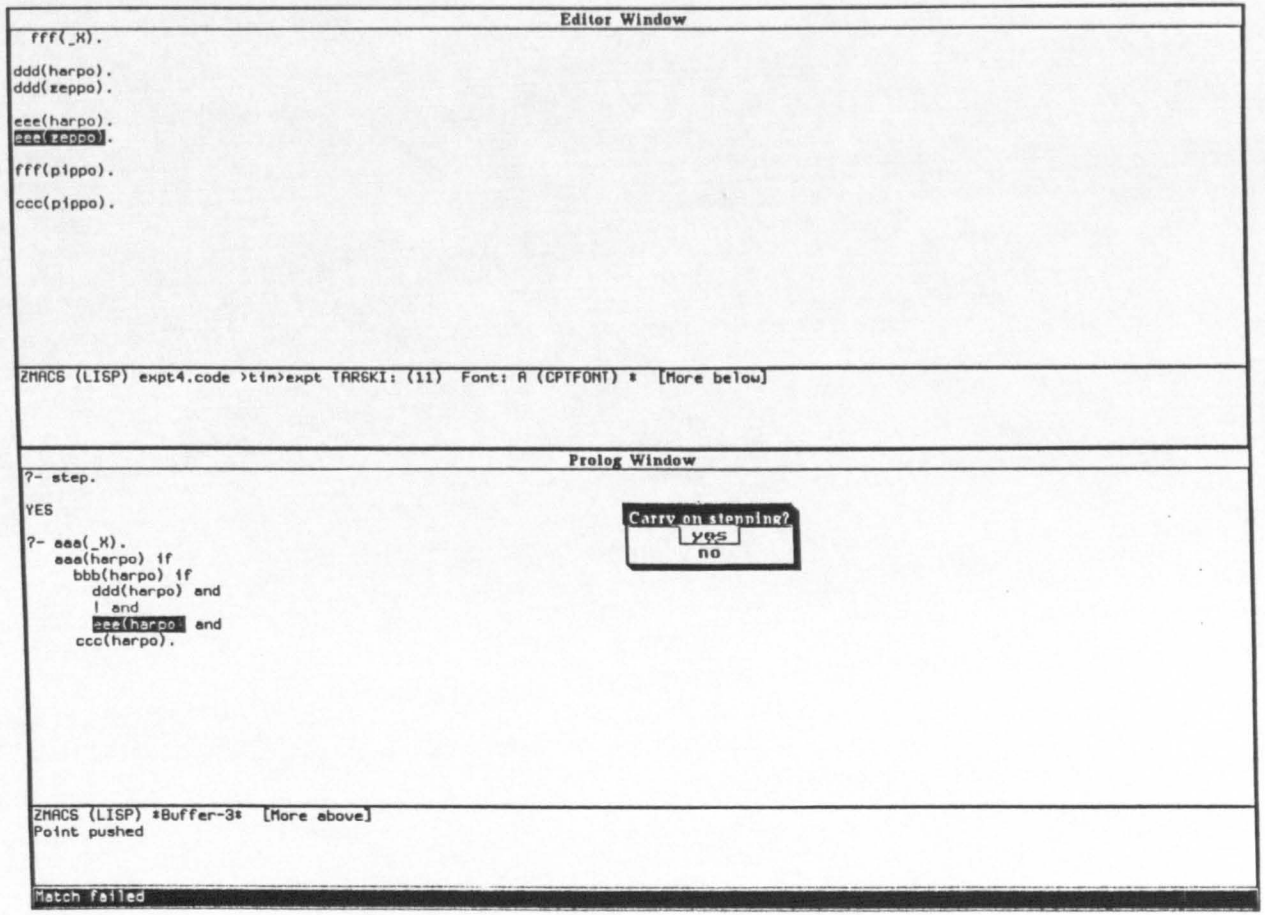
## APPENDIX-I

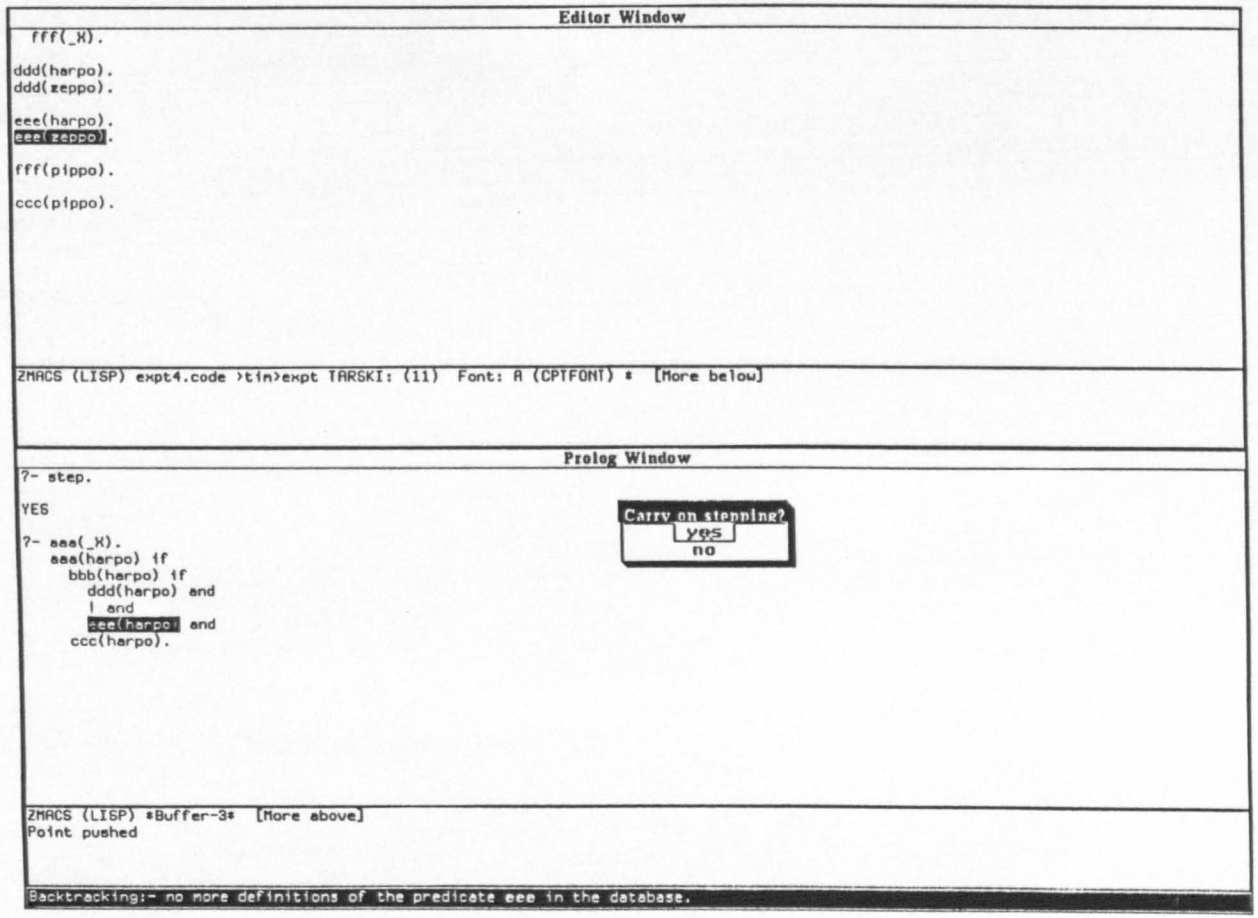
Appendix I presents an example of how APT displays the cut in program execution. The example consists of an extract from the execution of the Prolog program listed below.

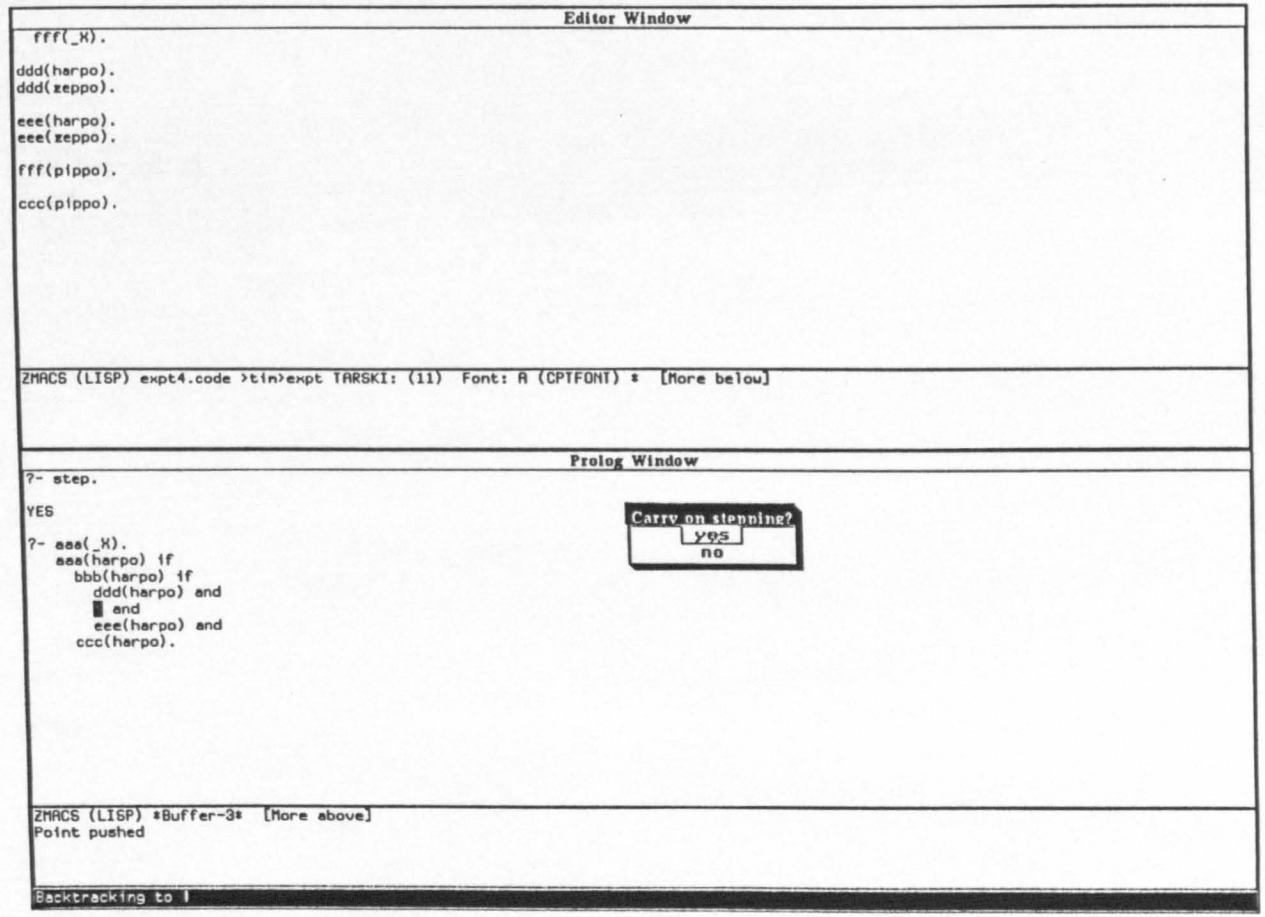
```
aaa(_X) if
  bbb(_X) and
  ccc(_X).

bbb(_X) if
  ddd(_X) and
  ! and
  eee(_X).
bbb(_X) if
  fff(_X).

ddd(harpo).
ddd(zeppo).

eee(harpo).
eee(zeppo).

fff(pippo).

ccc(pippo).
```

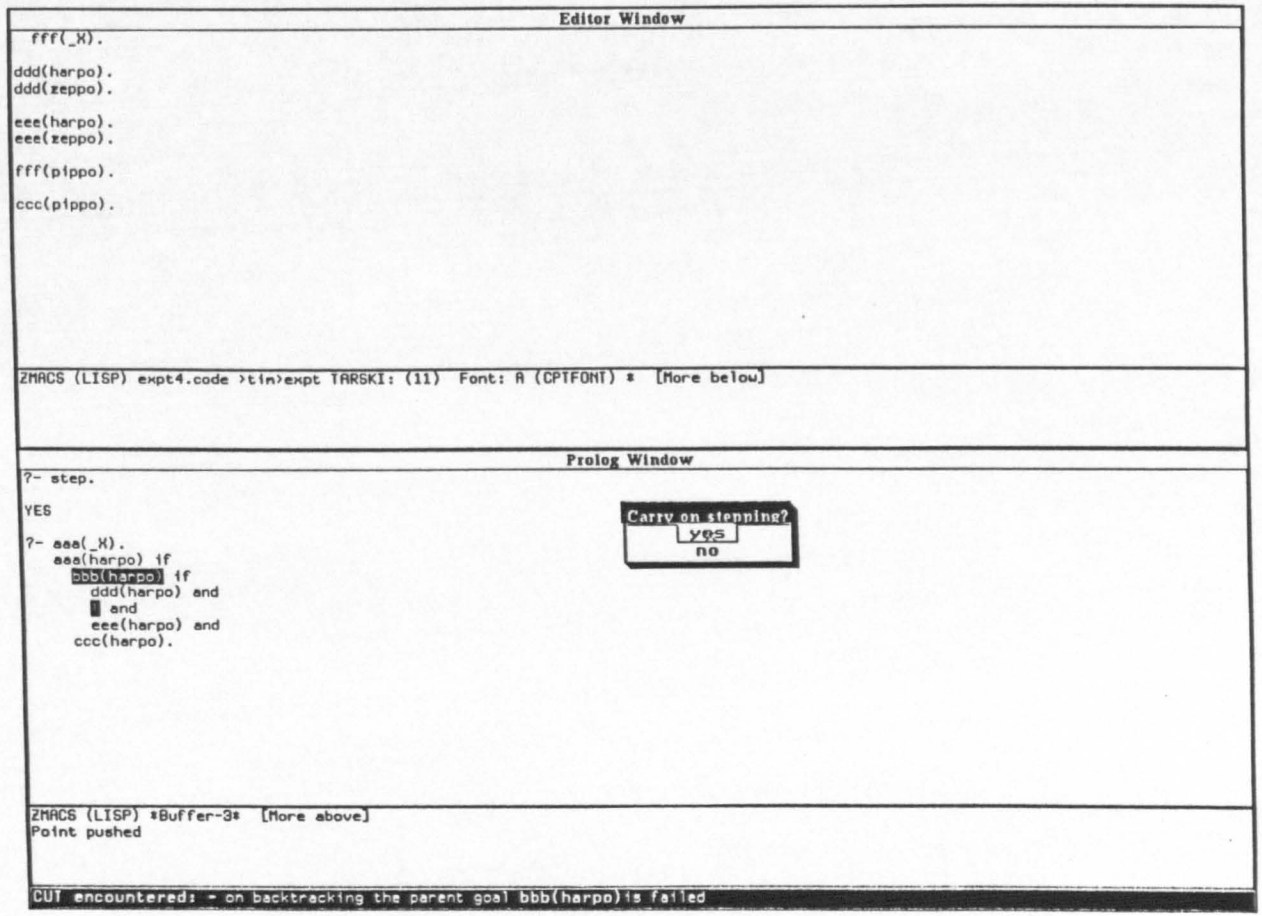

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tin>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(harpo) if
bbb(harpo) if
ddd(harpo) and
! and
eee(harpo) and
ccc(harpo).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal eee(harpo) against eee(zeppo)

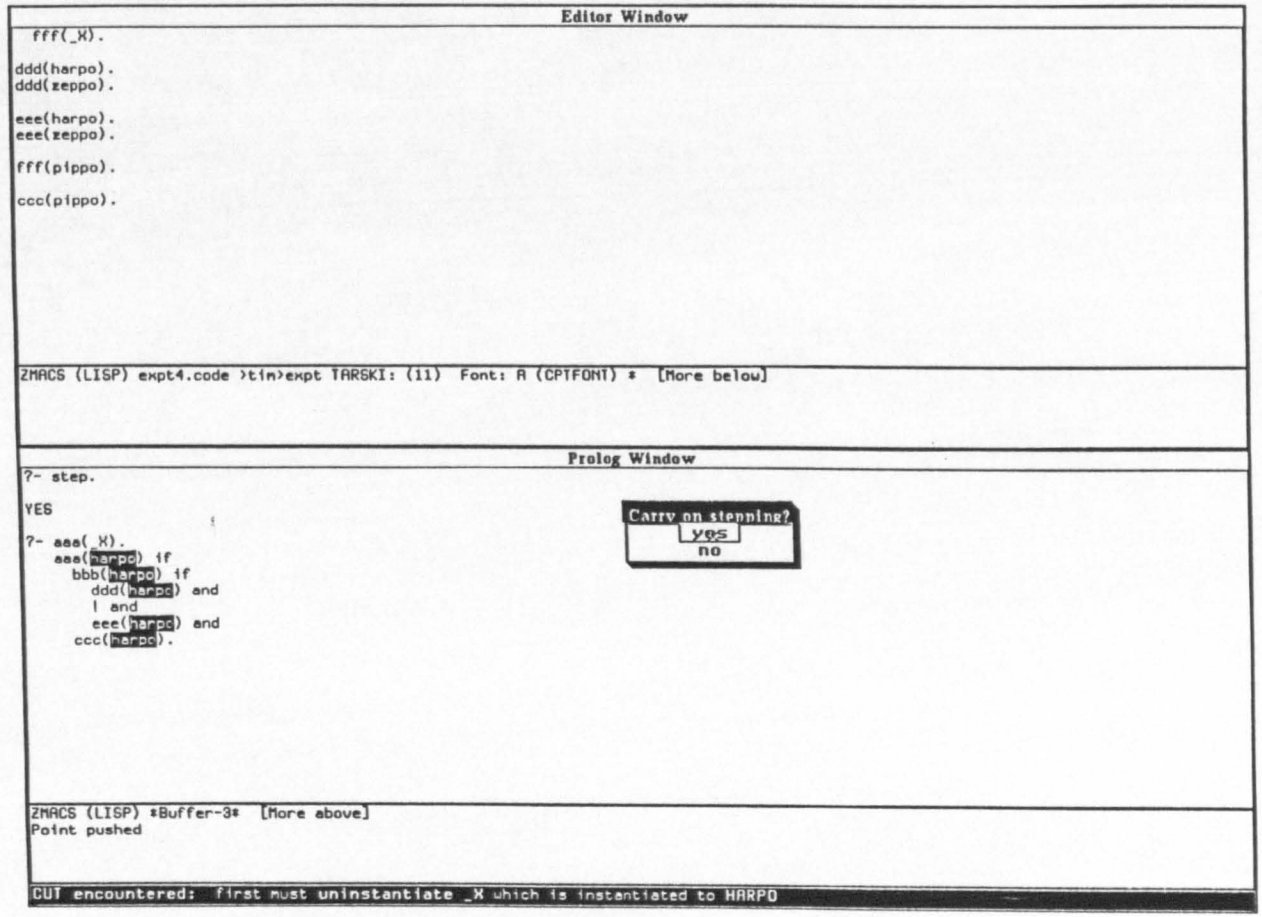

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tin>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(harpo) if
bbb(harpo) if
ddd(harpo) and
! and
eee(harpo) and
ccc(harpo).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Bactracking to eee(harpo)

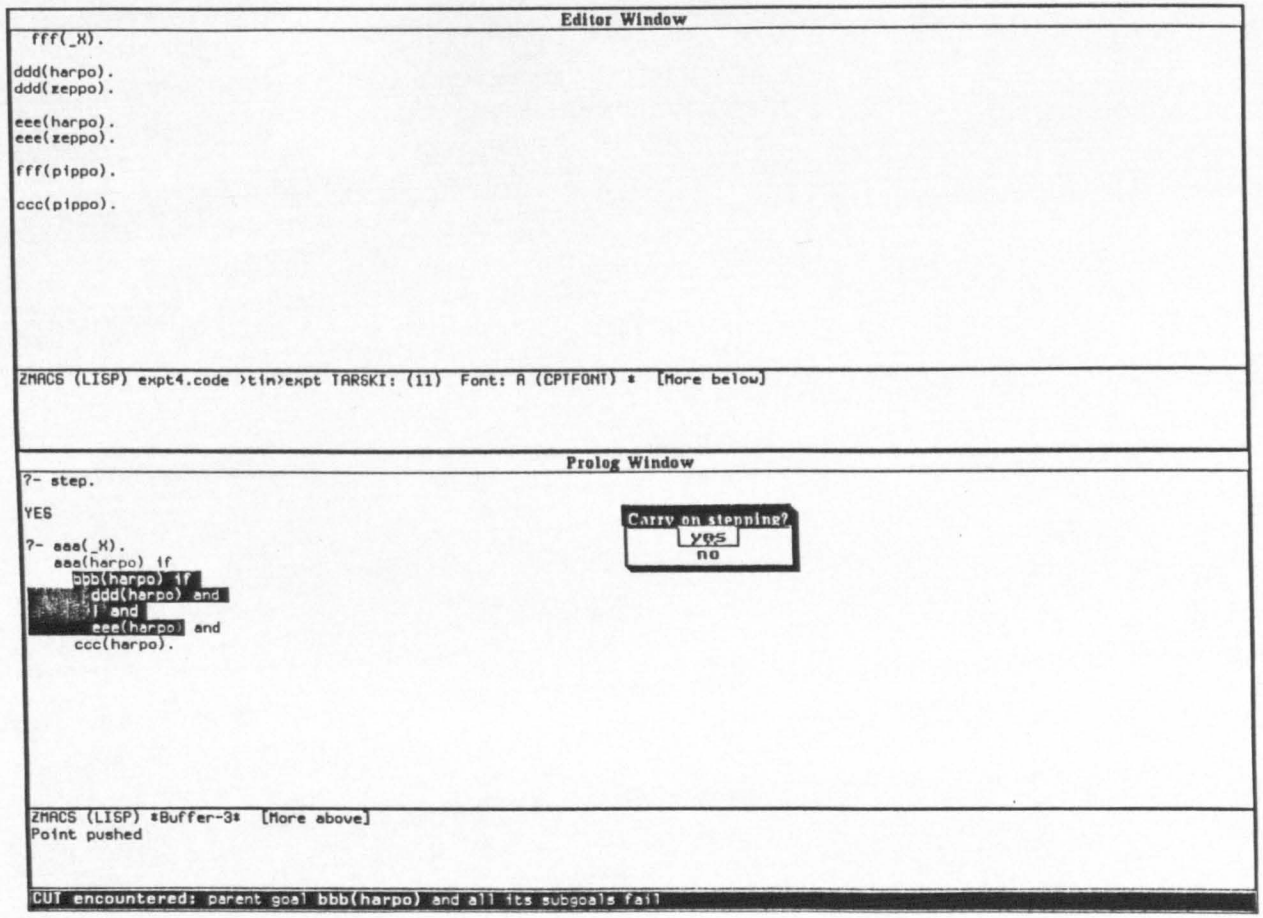

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tim>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(harpo) if
bbb(harpo) if
ddd(harpo) and
! and
eee(harpo) and
ccc(harpo).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed

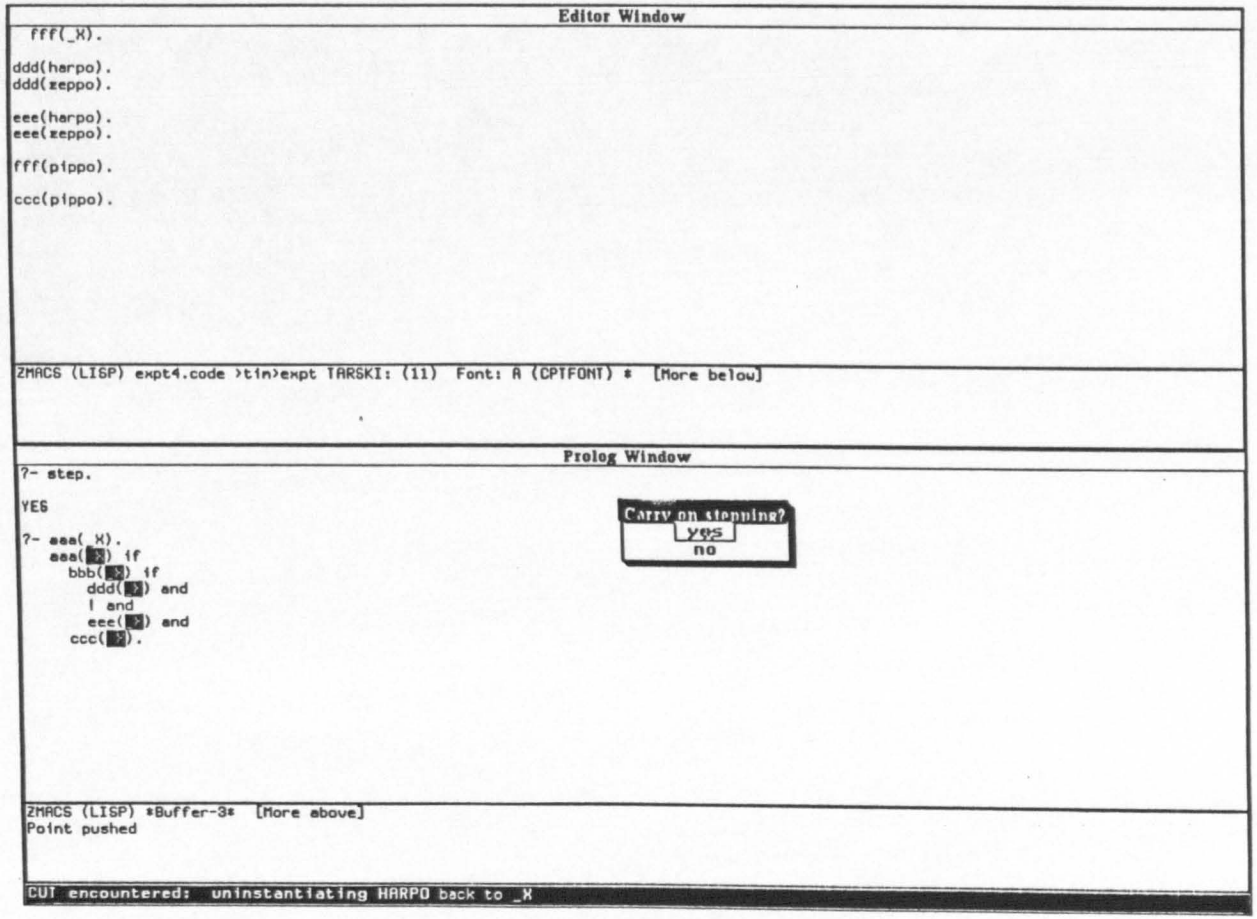

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tim>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(harpo) if
bbb(harpo) if
ddd(harpo) and
! and
eee(harpo) and
ccc(harpo).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- no more definitions of the predicate eee in the database.

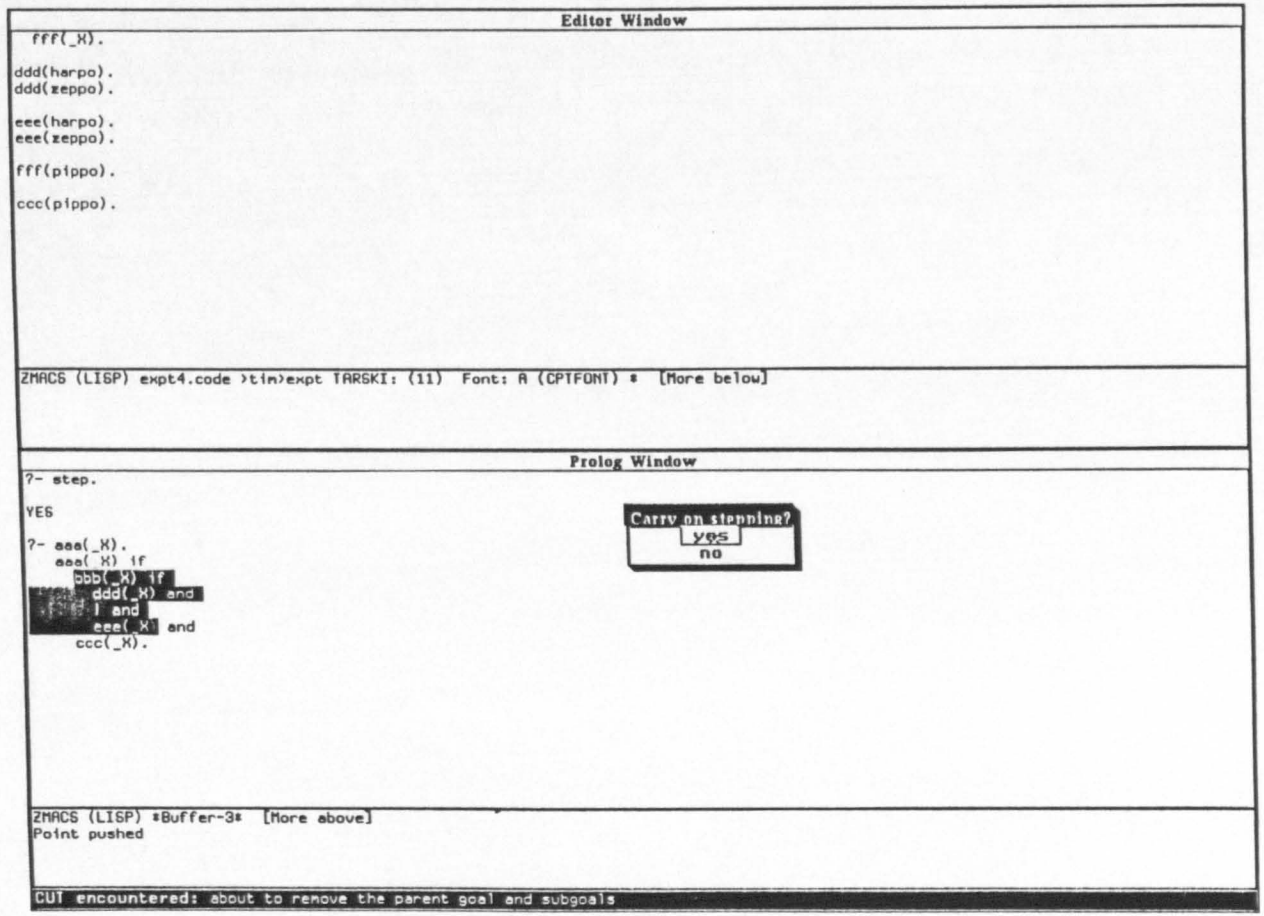

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tin>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(harpo) if
bbb(harpo) if
ddd(harpo) and
and
eee(harpo) and
ccc(harpo).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking to

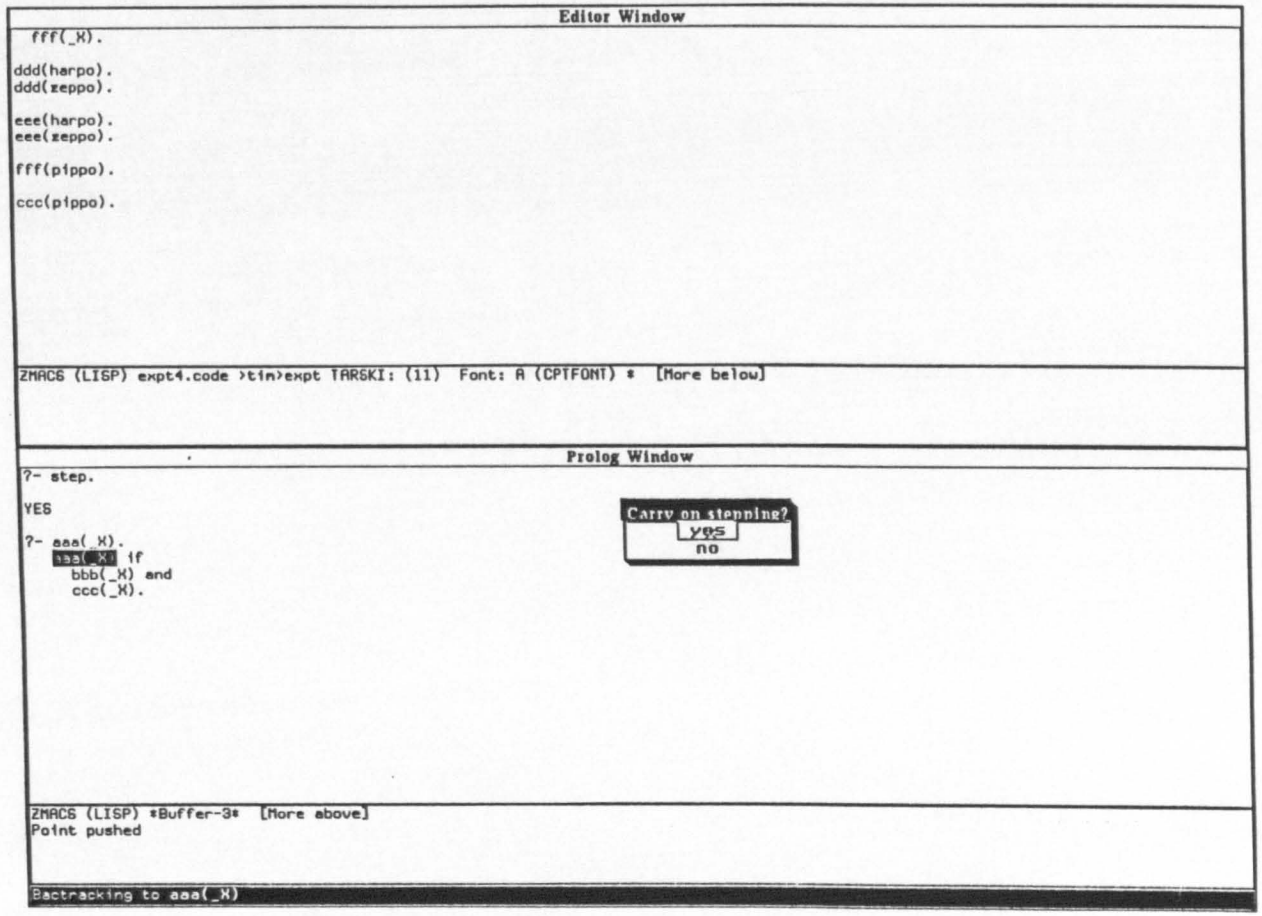

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tin>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(harpo) if
bbb(harpo) if
ddd(harpo) and
and
eee(harpo) and
ccc(harpo).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
CUT encountered: - on backtracking the parent goal bbb(harpo)is failed

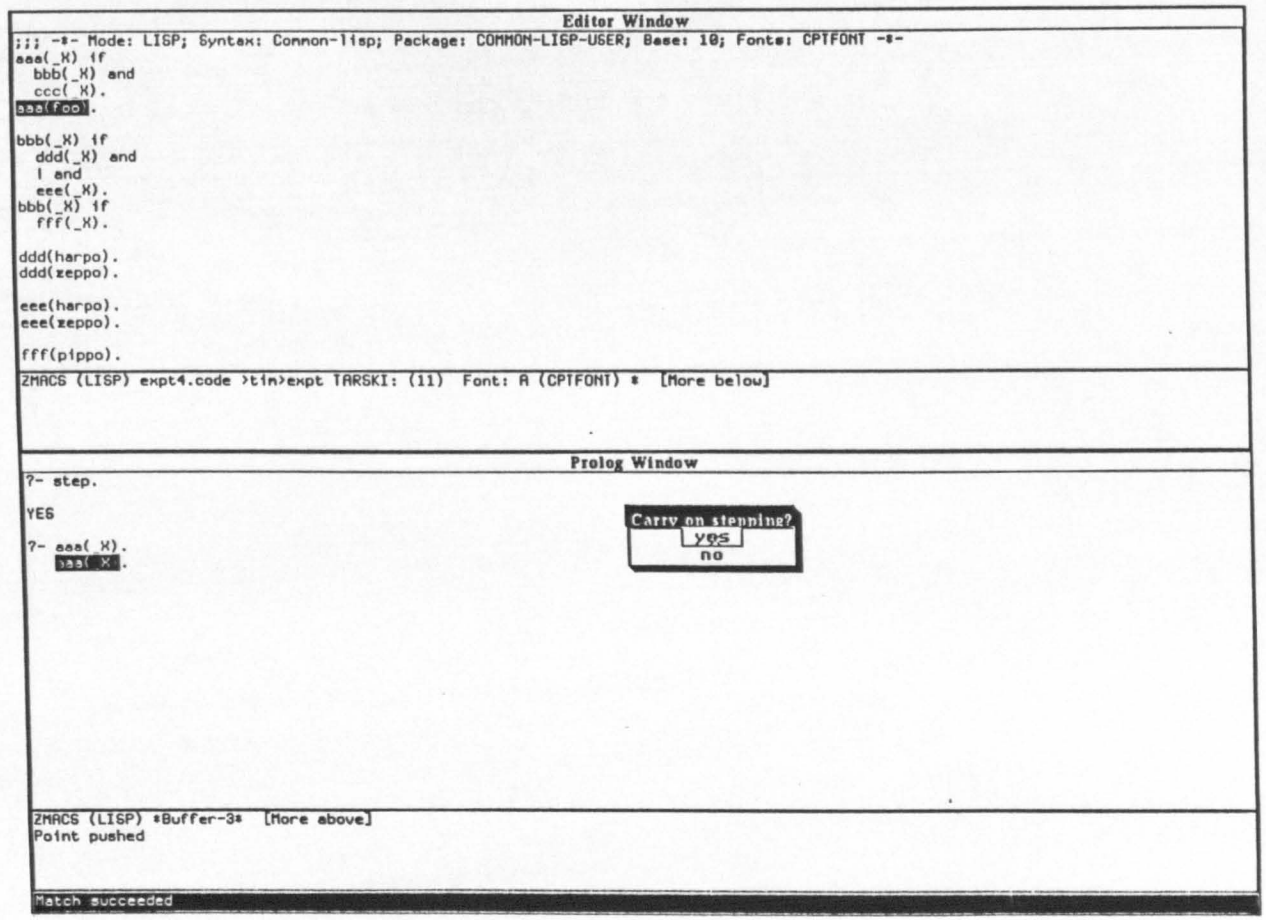

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tim>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(harpo) if
bbb(harpo) if
ddd(harpo) and
! and
eee(harpo) and
ccc(harpo).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
CUT encountered: first must uninstantiate _X which is instantiated to HARPO

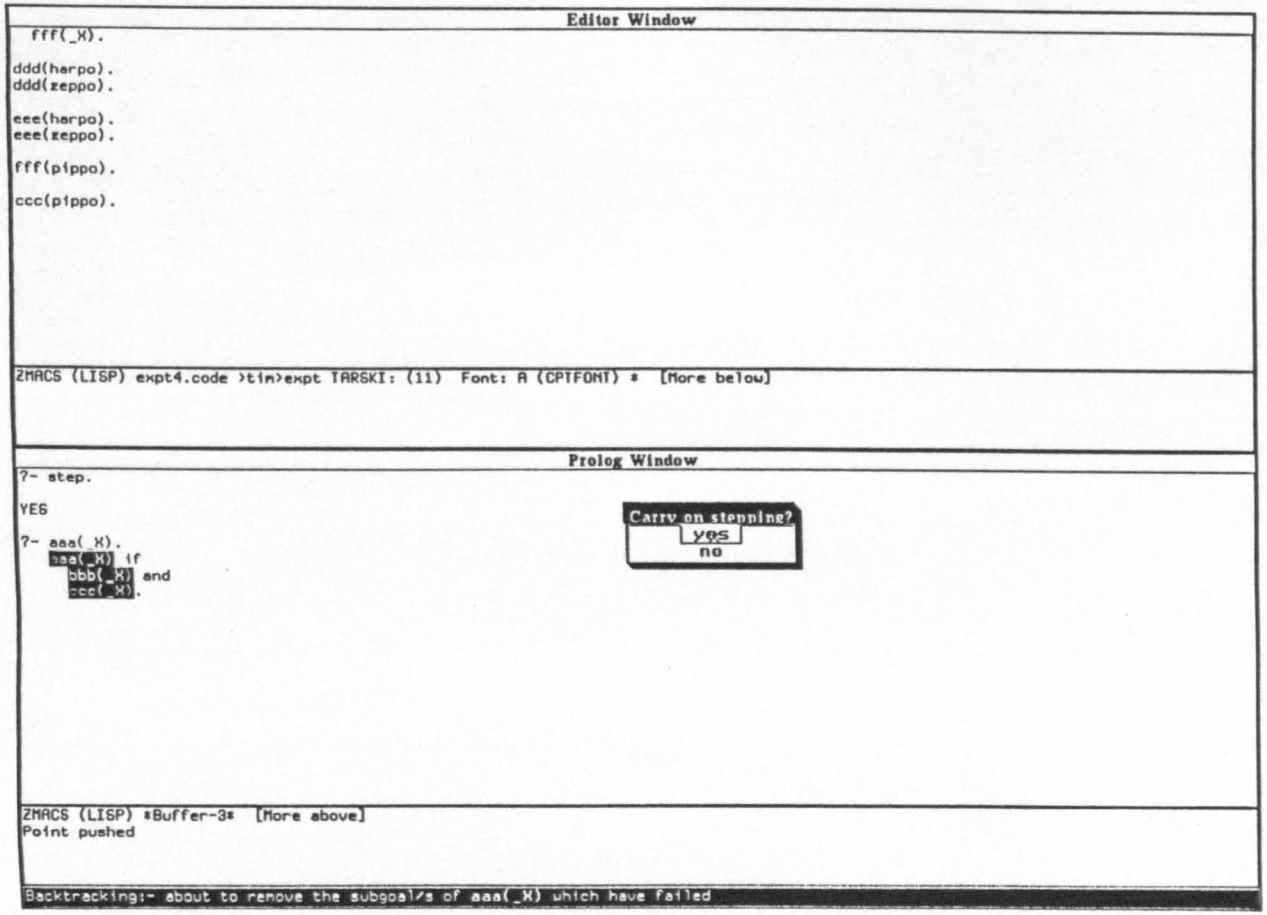

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tim>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(harpo) if
bbb(harpo) if
ddd(harpo) and
fff and
eee(harpo) and
ccc(harpo).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
CUT encountered: parent goal bbb(harpo) and all its subgoals fail

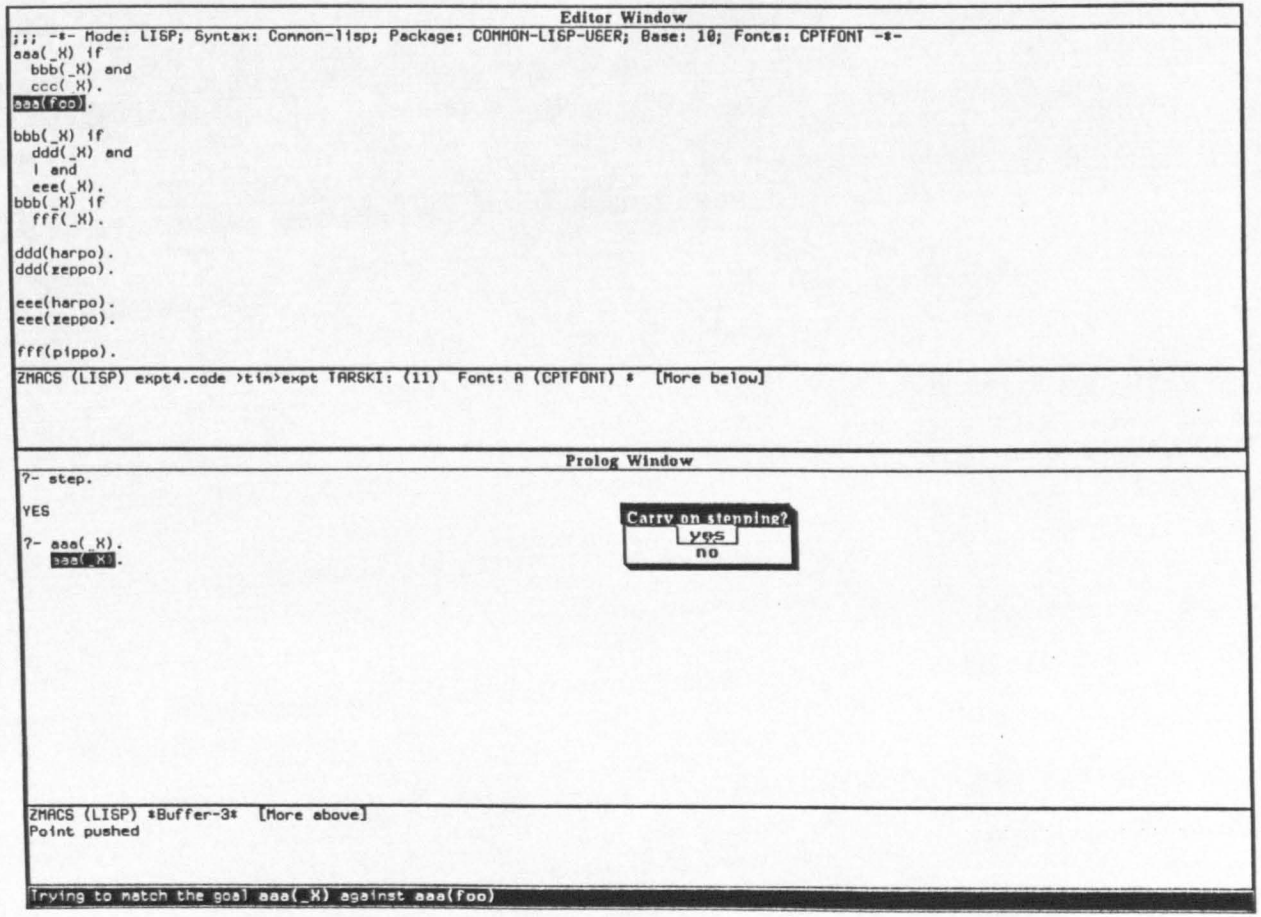

Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tin>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(_) if
bbb(_) if
ddd(_) and
! and
eee(_) and
ccc(_).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
CUT encountered: uninstantiating HARPO back to _X


Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tim>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(_X) if
bbb(_X) if
ddd(_X) and
l and
eee(_X) and
ccc(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
CUT encountered: about to remove the parent goal and subgoals


Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tim>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(_X) if
bbb(_X) and
ccc(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking to aaa(_X)


Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Font*: CPTFONT -*-
aaa(_X) if
bbb(_X) and
ccc(_X).
aaa(foo).
bbb(_X) if
ddd(_X) and
1 and
eee(_X).
bbb(_X) if
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ZMACS (LISP) expt4.code >tin>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa( X).
aaa( X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded


Editor Window
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ccc(pippo).
ZMACS (LISP) expt4.code >tin>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(_X) if
bbb(_X) and
ccc(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Backtracking:- about to remove the subgoal/s of aaa(_X) which have failed


Editor Window
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10; Fonts: CPTFONT -*-
aaa(_X) if
bbb(_X) and
ccc(_X).
aaa(foo).
bbb(_X) if
ddd(_X) and
! and
eee(_X).
bbb(_X) if
fff(_X).
ddd(harpo).
ddd(zeppo).
eee(harpo).
eee(zeppo).
fff(pippo).
ZMACS (LISP) expt4.code >tim>expt TARSKI: (11) Font: A (CPTFONT) * [More below]
Prolog Window
?- step.
YES
?- aaa(_X).
aaa(_X).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal aaa(_X) against aaa(foo)

# APPENDIX-J

Appendix J presents the transcribed protocols of the subjects (see initials for identification purposes) as they stepped through the prototype displays. The corresponding experiment is described in chapter 5.

## IP's Protocol for the Prolog APT-0

*IP: Is the word interaction of any consequence, does it mean anything? Other than being a label.*

EX: Its just a label the same as database.

*IP: Yes. So each time the program comes up you want me to explain what I think the program is doing, what its for.*

EX: Yes. Its going to be the same program as this. What I want you to do first is to look at the program and tell me if you can understand it, what you think its meant to do, running through it line by line. What that would actually be called with, what you would say is 'has_flu(June).' That is the program. You'd load that into the database and then at the bottom in the Prolog window you'd say 'has_flu(June).'

*IP: Well it looks to me as though its a bit peculiar. Er it looks like.. is it Mary kisses John, and John kisses June. Is that right .. yes.. and then it says if X, sorry X has flu if X kisses Y and Y has flu. And then it says Mary has flu. So presumably you're going to find out em then that would say that John has flu cause he kisses Mary.*

EX: Yes carry on, I want you to tell me..

*IP: Ah right. Because Johns been kissing Mary and June kisses John therefore June will get flu as well. Plus anybody else not mentioned her'e. Thats all I see from that.*

EX: O.K. You might like to reread the introduction sheet. I'm not going to tell you too much about the tracer as its meant to be as obvious as possible. The sought of things I want you to say is if anything confuses you mention it, and anything that comes to mind, and whats going on with the program, what you think its doing at each frame. And right at the end I'll ask you for your comments. The 'f' and the 'b' are forwards and backwards and you shouldn't need to use the 's'.

*IP: So this is the same thats on here, so I want to go forwards now as I've already explained whats on here. Matching head... oh right. So its looking for an X, sorry its looking for the beginning of the rule which is has_flu (X). Now if I press f I imagine..*

*would I say it would have done that.*

*Yes I would have thought that it would go for an X, which I suppose it is doing for as it says kisses Y X, Y and X...*

*and it's matched the X with the has_flu X of kisses flu X and its going to look for a match between, looking for something that matches with that X. And its got June why has it go June oh I see. I would have thought it would have gone for John myself but anyway its gone for June.*

*X is instantiated to June*

*kisses Y June so thats June its looking for now so its going to, well its going to match John isnt it*

*Yes in actual fact*

*Now its looking for the Y and having matched that I would say it will instantiate I suppose has_flu Y will be has_flu John*

*Yes I was right.*

*Did I miss something out. Yes a clause it seems kisses John June.*

*has_flu John*

*Matching head did I miss something. I'm going to go backwards*

*has_flu John*

*so John has flu if he kisses Y and Y has flu so its going to look for another person thats been kissed*

*So John has flu has been put in there. Kisses John June Yes. Shall I just go back a bit Trying rule .. that what it did before*

*but its got a different answer this time.. which is being instantiated in the interaction*

*So it will be looking for a match now between John kissing someone which will be June*

*wrong been looking at the wrong bit there*

*thats right so it puts the Y where the Mary would be*

*so now the Mary is going to go into the has_flu Y, Y being Mary*

*Yes I see. It seems that when I describe it I describe 3 things at the same time, at least 3 things on the screen here. Happening at the same time .. at least I assume I do, maybe its because I'm too lazy to describe them at each detailed bit*

EX: What bits are you describing?

*IP: When I say that, let me go back when I say that the Y is matched to Mary.. Mary kissing John than i say that Mary has_flu*

*thats the Y is matched to Mary*

*that's when i don't say that it looks for the Y. I just say that Y goes straight to Mary.*

EX: O.K.

*IP: like that Y is instantiated to Mary so Mary has flu .. thats what it says on the bottom line...*

*that Y clause succeeds*

*has-flu mary*

*Matching head. So its started again now has it.. Yes.. has flu Mary. So does Mary have flu, now looking for oh I see because the bottom of this rule, the tail*

*of this rule is the head of the rule. I suppose that was what was going on in the beginning, thats what I didn't cotton on to. So it tries the whole rule*

*and it will be looking for something in the head of the rule that matches with the tail and its found.. that was Mary has flu being ..how has it found it, why does it go to that one kisses Mary.... has flu June, I suppose that was the original question I've forgotton now no it wasnt the original question*

*has flu Mary*

*right I see so has Mary got flu. Mary has flu if she kisses someone who has flu..so who is this someone.. Mary is the someone.*

*well quite by chance I've noticed that you've spelled instantiated wrong.*

EX: Oh yes.

*IP: Ehm! so Mary at the top there in the database is being matched with the Mary down here*

*so now its going to try the first part of the rule has kisses Y Mary look for a Mary, a match in the rest of the database for kisses Y Mary, which it won't find I say confidently this time*

*Oh no it doesn't so then its going to look having done that it will go to the next rule which is has_flu Y which will look to se whether Mary has flu and I happen to Know that it does have flu ...in another part of this rule*

*so I've missed out describing the start of backtracking by saying what its going to do next. But I still think I'm right*

*No I'm wrong*

*ha fails so it starts backtracking like it said*

*..... which means that it goes back to where it started from*

*has-flu Mary fails. Oh I see so fails on that count but because its backtracking I imagine its going to try on the second part of the rule which is has_flu Y*

*Remove failed code dont know what that means.*

*Right*

*and then it does what I said it would do retry has_flu Mary Yes*

*and its found a match*

*so the clause succeeds*

*trace ended*

*so thats the end of it is it*

EX: So any immediate comments

*IP: Well as I say when I was describing it I wasn't I found I wasn't always looking at the right bit. I found that most of the time I was looking at it I was focusing on*

*the word interaction, rather than what was going on I suppose it could be because it is in the middle of the screen.*

*And as it did things by describing what it was doing I found that I was describing like I said at least two lots of screen action at the same time so that I felt that I was saying things when nothing was happening or rather that*

EX: You were describing things all at one go.

*IP: Thats right. And I wasnt sure right from the beginning why it went and matched you know how it chose the first ones to match..*

EX: What has_flu(June).

*IP: Yes*

EX: Thats what you ask it to do. Thats what you type in. Thats why you have the question mark.

IP: Ah yes I didn't notice that.

EX: One thing that you seem to have confused is that.. these things.. its not a statement its a question.. like.. these have all been filled in with instantiations now but kisses Y June, so its asking does anyone 'Y' kiss June, and similarly when this gets filled in to John its asking has John got flu, its not saying John has got flu, because you don't know that yet, you can only say he does if these other things happen.

*IP: So all through this here its looking for something*

EX: That's right ....

EX: What did you think of the presentation?

*IP: Well I liked the way it showed, this bit of highlighting here showing the way it goes through because when I have seen Prolog doing a trace it zips through and its like double Dutch to me, not double Dutch if I look at each line but because of the way its produceed so quickly I found it almost impossible to follow where its gone wrong, it says fail, or its in a loop when you get the same pattern. but here I .. by showing the whole rule at the same time rather than just the one line that the trace shows its much clearer showing where the machine is going in its thinking bit. ....*

EX: The interaction is supposed to show as closely as possible whats actully happening when the code is run. How closely did that match the way you thought Prolog worked.

*IP: Well nearly.. the only thing it really caught me out on was when the backtracking started and it removed existing code now whether I knew that or whether I'd forgotten that I don't know.*

EX: That actually wouldn't happen in prolog, it only occurs in the trace because you cant leave it there on the trace, it would get in the way of the presentation..... so Ive got to take it away .... Did you like the idea of having small messages at the bottom, did they help.

*IP: Well yes they did because especially when I went back and they reminded me what I should have said, describing what was happening, so that was useful, and it more clearly showed what was going on when the matching was going on.*

EX: What about the level of detail, as you said that you decribed one thing happening for evey two or three on screen. Was there too much detail in the display?

*IP: No I don't think so it was just my laziness in describing what was happening, cause in analysis I probably wasn't describing things as accuratly as was going on. I was actually making a jump without realising it. I was missing things out in between whoch could have been a basic misunderstanding of what was happening. ....*

EX: Where ther any points in the program that were clarified by the stepper.

*IP: Not particularly, well I suppose the backtracking.... ....*

AE's Protocol for the Prolog APT-0

EX: The prompt in prolog is the ?, and you've just typed in has flu june, which is youre top level call, and er it should be self explanatory. As youve done no prolog i will help you if you get stuck.

*AE: Right ok, so its trying to match has flu june and its got has flu x, presumably its going to substitute june for x*

*right well, its em its expanded the definition, and now its going to match june to x*

*yes*

*so now it will see if June kisses anyone*

*which she does*

*so it will now substitute Y for John*

*em right so now we have has flu john and presumably its going to no we've got kisses John June so is it going to try and match that, let's go back*

*it's already matched that*

EX: It's trying it on for size, if you look at the comments

*AE: Oh right so that just saying that the clause has succeeded*

*so now its going to see if John has flu*

*it's found has flu which is recursive so we're going to do it again*

*substituting John for X*

*John kisses Mary*

*substitutes Mary for Y*

*succeeded*

*now its got to find out if Mary has flu*

*here we go again*

*see if Mary kisses anyone*

*and no match*

*so clause fails and it tells me its going to start backtracking*

*so it throws away that*

*er code on the failed*

*and has another attempt to match has flu*

*and its found it*

*succeeded*

*so we have a result yes*

*oh and thats the end*

EX: Any comments about that

*AE: Em*

EX: For instance I don't know whether you've seen a prolog trace that you get normally

*AE: No no*

EX: Em it tends to be of the form, its got keywords like trying this, failed, succeeded, the spy package

*AE: Oh I have seen it, goals and everything, yes I have seen that sort of thing*

EX: Do you think you would understand that with that tracer

*AE: This would be much more understandable, em .. I think having already got some concept of bactracking, what was happening there may of helped, wheras if I hadn't come across it before I might have been confused, em if there had been a, I don't know whether it might have. It might have been nice to see, no no. I was going to say in terms of learining about prolog if I'd seen one where it had failed and backtracked and then succeeded, but that would just depend on the particular example.*

EX: That is what that did, its tried the first has_flu and failed, it hasn't backtracked very far really, its backtracked to has_flu again. It got down to here and failed.

*AE: Yes ok*

EX: A nice idea may be to have a help facility, using keywords at the bottom, so you can say things like help backtracking, and it would give you a little demonstration with some text.

*AE: Yeh yeh*

IP's Protocol for the Lisp APT-0

EX: What do you think the program is doing first of all.

*IP: First of all its defining a function infect and em its got in it. Well the first one says x has flu and then there is a condition, and I read it as x has flu if and then if the following applies. This one is I'm not sure the first one says equals is that equals nil.*

EX: Yes

*IP: Get x kisses so thats find out whose been kissing, whose been kissing x and the answer to that is, if the answer to that is nil and then if it isnt true then somebody has been kissing x then you find out em whose been kissing x again. Same question again. and looks like it does the same thing again in fact....*

EX: So you're not really quite sure what it does

*IP: No not quite sure*

EX: But youve got some idea of some of the things it might be doing

*IP: Yes exactly*

EX: Right so if you remember that putprops put things into the database and get gets things out. If you load this into youre environment what actually happens is that things like this will get done immediatly. So if you load that in, if you load putprop 'mary 'john 'kisses into the environment what it actually does is put that into the database. So you have mary john kisses in the database. Whereas this doesnt get done because its inside a procedure, this only gets done whent the procedure is called.

*IP: The database at the top that repeats this so I don't have to look at that any longer. So Infect mary thats calling the function and I have to go forward now so I press f and*

*here it outlines the whole of the function infect x it says the called Lisp code so it refers back to whats in the database em so lets see now whats going to happen by going forwards.*

*and because its called it its taken it from the database and put it in the bit I'm working in at the moment I suppose to the interaction but it hasnt yet done anything with it. and now its taken the first part which the infect x and it will try and match x with mary*

*x is a variable*

*and x is mary. Oh I was right, so it matches, puts mary in there now I suppose what it will do now is put mary into the next x. Putprop x flu has*

*which indeed its about to do or is it. Unless x is something else other than mary.*

*x is a variable*

*so now I think it will instantiate mary to, yes putprop so now that is put into eh. Putprop will be put into the database will it. Since it's or is it because its part of a function it wouldnt be.*

*well let's see. Putprop mary flu has*

*so it has been put into the database yes*

*so what's it going to do next. It will try the condition and I should think it will go for the get x kisses as that is the middle most part of that condition.*

*well it does the whole condition first.*

*Oh mary has .... did I say that.. no I didnt say that so its gone to equals get x kisses nil rather than get x kisses. conditional the test so what does that mean. finding out who x is grom the kisses database.. and see whether it equals nil. I dont know what eq nil would be.*

*Trying to instantiate the variable here*

*and it matches with mary oh thats clever so x is instantited to mary. So get mary kisses, so it will try and get mary kisses from the database which wont succeed because there isnt a mary kisses up there in the database. Oh yes there is. putprop mary john kisses.*

*Put mary kisses. So thats the one that it matches with. mary john kisses so I suppose now it will try the put mary it will be equal mary nil, putting the get mary kisses into the equal nil clause.*

*so it's got mary kisses why should it want john where is there any room for john to go into the equation really.*

*it's assuming its a condition*

*something about equals john nil*

*get mary kisses gives you thats the evaluation so its passing it to, passing john to equal*

*mary john*

*to another part of the property*

EX: Don't forget what get does

*IP: It's taking information out of the database*

EX: You've already got get mary kisses so what is the new information

*IP: That who she kisses is john so That's passed to the equals nil*

EX: Remember what sexpressions do they return values, so the sexpression get mary kisses returns the value of john.

*IP: Yeh*

*so equals john nil. I dont think that john does equal nil.*

EX: Don't forget the comments at the bottom

*IP: Conditional evaluation of test the test is still going on then. What did I say*

*equal john nil oh yes. so if it returns the result there which is nil so*

*it returns nil so It goes on to the next bit. Is t for test or t for true here.*

EX: Look at the status line

*IP: It says the test so its still doing the test. evaluates to t. the test evaluates to t... because it was nil.*

EX: This is a new conditional statement. you can have more than one statement in a conditional remember. The last one you usually put t at the begining which is the test and that test is always true.

*IP: That catchall test at the end. that one so this one is always true. and this time it says get x kisses so again its going to be looking for something in the database. and em x could be anybody I*

*suppose it could be mary or john but as it's infect I suppose it's going to be mary. looking for the variable x*

*and mary, x is instantiated to mary because we have the infect the first part of the function. Infect x, get x. If it had been get Y down there it would have looked for somebody else or it would have failed. so because thats the same variable it has to be instantiated to mary when it first ran through. So get mary kisses its asking now which is exactly what it did before. so*

*get mary kisses so its going to look for mary kisses out of the database and it will find mary john kisses out of the database and since it has to return a value it will return john to the test er Infect the bottom line here.*

*there's john picked out*

*infect john and since that says thats always true,*

*infect john*

*it's looking at the called lisp cade as it says on the comment, but I didn't really look at the comment before that*

*none are thats why I didnt look at it*

*so now it will be passing, giving the answer john, is that right yes*

*putprop x flu has. Let's go back a bit here*

EX: Let me mention recursion

*IP: Em*

EX: Do you know what recursion is

*IP: I've read about it, but I havent done it in the big book yet. Its where em a function calls itself as opposed to recursion.*

EX: So you see whats happened here Infect has called itself from within itself.

*IP: So because its gone back up to the top line here. If I go back a bit*

*So its got john there being called*

EX: That's exactly the same as with Infect mary

*IP: Yes it's just called itself again there*

*so now its going to go through the whole routine again to find out em only with john*

*instantiated to x would it be*

*true infect john*

EX: x is a variable

*IP: Yes*

EX: john isnt a variable

*IP: No, sexpression evaluates to john I can see that, so it goes into the bottom of the condition yes*

*so now it starts all over again*

*yes by having infect john in the first part of the function which yes right putprop x flu has so now its going to have a look in the properties, no it puts em the property a new one into the database at this point. x flu has and x is a variable and we know that mary has flu so the x will match with mary ... no it matches with john*

EX: Do you see why

*IP: Because infect john .. matches, because it's doing it again its not going to ... but it's actually looking at john here so that x will match with john there.*

EX: First of all get x kisses x gets instantiated to john in the first time round. x from then on equals john until its changed.

*IP: Yes*

EX: It originally equaled mary but as soon as it becomes reset to something else it equals that.

*IP: So in here from this time in it is reset to john. So from now on there will be a new one with x instantiated to john.*

*so this putprop john flu has will also go into the database*

*putprop john flu has, yes that's another addition to the database, and then whats going to happen next*

*goes onto the condition the test the conditional part and this time it will be equal (get x kisses) nil thats what we're looking at*

*and it will do the get x kisses bit the x*

*which is the variable which is going to be john get john kisses that we're looking for in the database. x is instantiated to john*

*get john kisses so thats what its looking for now and its going to find june, the putprop john june kisses*

*so now it will return the answer june to the equal june nil test*

*june*

*equal june nil test, yes. so that means that will return the answer nil as well as the one above with mary equal june nil no it doesnt so it return the answer nil*

*so then it goes on to the one we know is true on the last line true infect get x kisses the test evaluates to true*

*get x kisses, so its going to look for x here and it will find john again*

*yes variable*

*and it's john because its the first of the test there*

*now it will be looking for the property get john kisses and which is done already and its found june ..*

*get john june kisses*

*june so that means that this test evaluates to june*

*infect june and its true that it infects june. so I imagine that now it will go.. why wouldn't it do it again*

*infect june*

*it goes up to the top with the call to the lisp code true infect june*

*it is going to do it again*

*so t infect june starts it off all over again*

*and this time its going to instantiate, it's going to do the putprop x flu has so its going to put june into the x of the x flu has*

*matches there and puts that into the property list*

*there is a new item, and now its going to do again equal get x kisses nil with june this time*

*so the test has just mentioned equal get x kisses nil*

*x is a variable so it will put june there*

*yes and again get x june kisses is what it will look for in the database but this time it won't find anybody because june is the one being kissed rather than the one doing the kissing*

*so it's looking at the database there to see if it can find a match with any of those*

*and it cant so it evaluates to nil so equals nil...*

*does equal nil nil, well it does equal nil*

*which is true*

*so condition is true, so it returns the answer nil*

*because of the condition*

*evaluation what does that mean.. evaluation is the same as what I've been saying as answer all along. so procedure evaluates to nil.*

*so thats the answer it should return*

*condition nil*

*true t nil. I don't know, I noticed before the brackets*

EX: Yes those brackets are ..

*IP: Are they pointers like arrows*

EX: There are actually part of the previous procedure, but it would be confusing to put them anywhere else.

*IP: So now we've said that its true that its nil in the last test, then the procedure evaluates to nil*

EX: All this is the action of the one above

*IP: So infects john is ..I think I've got lost somewhere I think*

*yes*

*oh I see that nill is sort of backed up again*

*and that procedure therefore evaluates to nil .. so thats the answer I think Infect mary*

*program has finished so thats the answer.*

*so the nil moves all the way back up all the way up the test all the way to the beginning*

EX: Well its not the same one really each thing is returning nil

*IP: Each time it fires it returns nil and it finishes once its got the third nil out of the database. because it didn't find, when it did the test it didn't find anyone in the database.*

EX: So you've not come across recursion before

*IP: no I've not seen it working, but i have read about it*

EX: Does that give you any idea how it works

*IP: Yes I think it does because it, the way the clause is written, I suppose it must be this bit repeating like that, that means that it goes through it over and over again until it comes across a nil.*

EX: Do you see what the program has done now

*IP: Yes I think so. Its, the final nil came up because there was no match in the database*

EX: What have you got in the database that you didn't have at the start

*IP: You've got the two putprops at the bottom there, er the three putprops mary flu has; john flu has; and june flu has.*

EX: So what have you done

*IP: Well its increased the database*

EX: Yes, but in terms of the program well its propergated the property flu through mary john and june, starting with you explicitly giving mary the flu by the first putprop. You actually say Infect mary

EX: Any comments about the display

*IP: The only time I didn't understand it, well it was not so much as it didn't do what I expected it to but em as before I would jump over steps where it was going slower than what I was saying but then sometimes it would do things that I hadn't said at all. I mean I hadn't assumed them it just went to them.*

EX: Do you think that if it misses out those bits you would not have had as clear an idea of how the program worked.

*IP: Yes, I think it was important that all the detail was there, because although I say these things and make 3 steps in 1 go I'm not sure how I get there. especially being able to go backwards you could see more clearly what was going on.*

EX: What about the highlighting and the display of information did that seem sensible

*IP: Oh yes*

EX: And what about the messages

*IP: The line down here, that was useful as well.*

*I think that it was always clear what was going on, although not always clear to me because er*

EX: If you had been using this with a book this would have clarified things in the book?

*IP: Oh yes I'm sure it would, if I'd had this from the beginning for the simpler things.*

EX: So you think that using it for demonstrating things would help

*IP: Yes. I would find instantiation of variables alright, but I think for recursion and if you could show iteration as well seperatly and you could show the difference between the two then that would be better still. Since as soon as I get one in my head the other one goes out.*

EX: What about conditional clauses, because you were having trouble with them yesterday weren't you.

*IP: Yes. that was because I had misread what it said in the book about it returning that nil there, that result. Ithought it would return the answer for get x kisses*

EX: That is still part of the test, you have to evaluate all the fields before you get the answer.

*IP: One thing that confused me first time round was that I thought that it would do the get x kisses first whereas you highlighted equal get x kisses nil.*

EX: That was supossed to be more informative showing the conditional, then the test, and then it saying the first bit you evaluate is get x kisses.

*IP: Because I wasn't expecting that and then it went on to do what I was expecting it to do, and pass that value to the equals nil, so I just always think it goes to the middle bit first and then works its way out.*

If the test had been more complicated I might not have known where the test was.

AE's Protocol for the Lisp APT-0

EX: First of all what I want you to do is to run through this program and tell me what you think about it

*AE: Ahm right well its defining a function called infect which has an argument x er..*

EX: Just say generally what your thinking

*AE: Em.. I presume putprop is going to put something into a database. so the first thing is that when you infect x , x is given the property that x has flu and then err... if er... if x kisses something then something also gets flu... em*

EX: So what's the general idea of the program

*AE: Well your passing the property of having flu from em x to mary*

*AE: O.K.*

EX: Right read the instructions

*AE: So the interaction says infect mary so presumably this is going to have an affect on anyone who mary has kissed.*

EX: That's like you would have typed that in to your top-level

*AE: If I go forward now then*

*right so its just filling in the definition of that function*

*so its going to match mary to x now presumably*

*great*

*and now its looking at the proposition that x has flu, and presumably that will become mary.*

*yes*

*so it puts in that proposition into the database*

*now its doing the conditional*

*er looking for something that matches er.. get x kisses*

*so its going to find what x is, in this case mary*

*er now presumably its going to look in the database and find out if mary kisses anyone which she does kisses john { 1 }*

*so fills in john for x { 3c }*

*and it's now testing whether john is nil. I'm not quite sure why em... ..... I think I'll go back*

*right*

*it's looking for something that matches mary kisses and it matches john em I guess what I don't understand is the original program actually*

EX: What actually happens with get x kisses it returns the value of x

*AE: Yes which was john*

EX: So john has replaced get x kisses. The other sexpression was equals whatever get x kisses

*AE: Yes I think what I don't understand is the cond. em*

EX: The first ones like the exit clause, if condition is nil, if there is no one there then as it says there the action is nil. thats what it returns. The effect of that is to bottom out the recursion.

*AE: Right so this first action is nil*

EX: That's part of the test get x kisses equals nil, the nil is the action

*AE: I see and if it does get a match then it will do the other one*

EX: It will only do this action if the first part the test evaluates to t, if its true. if its not true then it just goes on to the next conditional statement.

*AE: Right ok so*

*so now its evaluated to true*

*so now its trying to match infect x kisses so trying to match x which is mary and lets see who mary kisses*

*and she kissed john*

*so john's infected*

*Ah right so I see yes em there is a recursion here that I didn't notice, didn't register it the first time. Mary's kissed john and it's infect again well see who he infects.*

*going through it again*

*now johns got flu see who john kisses*

*in the database to see if john's kissed anyone*

*yes he has and its june*

*it's found june and its doing the cond*

*which evaluates to true again*

*so another call to infect*

*so well see who june kisses*

*ah it's just substituting john for x*

*so we see john kisses*

*he kisses june*

*and call infect june*

*much the same again*

*and june kisses isn't in the database*

*so this time we've got eh, we drop out of the cond*

*which evaluates to nil*

*so now we are coming out of the recursion*

*and we've finished*

EX: First of all have you got any comments

*AE: Well the obvious thing was that I missed when reading through it was that there was a recursion in it, and the other thing was not understanding how the cond worked*

EX: Did the display help

*AE: I think with a bit more time I would have worked out what cond was doing without your explanation*

EX: By going through the trace

*AE: Yes backwards and forwards*

EX: The way that it showed get june kisses wasn't in the database was that clear

*AE: Em it could have been clearer, I guessed what was happening but I actually went back and looked myself. I might have been confused by the fact that there was a john june kisses, but I realised that was different*

EX: What about at the end when you wind back up the recursion

*AE: I was, I realised what was happening winding up the recursion, the highlight of the parenthasees, I wasn't quite sure why that was happening.*

EX: That was indicating the extent of that function where it started and finished

*AE: Oh of course*

EX: What things did you like about the display

*AE: Em the way that the when you do the next recursive call, when the code goes in down there, it makes it clear whats going on, and conversely when you come back up the recursion. that sought of fits in with my idea of recursive functioning, which took a long time to build up, but the idea of going down and coming back up again.*

EX: Any things you didn't like

*AE: No there was nothing else*

AE's Protocol for the Assembler APT-0

*AE: Right do you want me to go through this*

EX: Yes, you don't have to go through it in detail. what its doing is that its got two sts of numbers, which are sequentially addressed and its adding them together, and storing them

*AE: Yeh..... OK.... I think*

EX: This is just initialising

*AE: Yes and then it loops round*

*AE: That's just a message for me, I go forward now*

*right*

*there just setting up constants*

*SED was, is that just the entry point*

EX: That actually sets the decimal, but this is the first part its going through dealing with constants

*AE: I see so its doing, Oh right*

*it's actually going through the whole thing doing the substitutions, oh right*

*em I wasn't , yeh. I wasn't sure what that meant in the instructions when you said mnemonics I assumed you meant the*

EX: It's difficult to think of a word, they aren't really variables

*AE: I wasn't to clear what you were going to do, I thought surely he's not going to put in the machine code values*

EX: That has been suggested

*AE: Well some weirdos like that sort of thing*

*it actually evaluates the constants*

*right execution*

*remove assignment mnemonics, ah I see*

*and there is memory, and the registers*

EX: That's setting the decimal, so we can work in decimal

*AE: OK so it just puts the bit in the register*

*and its stuck a 3 in there*

*store that in memory*

EX: Everything happening as you would expect

*AE: Yeh*

*so now its set up those little bits of memory, over there where the constants where*

*now we load Y*

*constant 2*

*clear carry*

*right so youre sort of using the convention that a blank is equivalent to undefined, carry had a blank in now it has a 0*

EX: Yes, em again there is a problem there of whether you stick random rubbish in there or what

*AE: Oh I see so it displays what the equivalent instruction on the bottom line here of what LOAD A 10, Y evaluates to now that Y actully has a value in it*

*but it doesn't show on there, fair enough*

*so it's 12 so it looks in address 12*

*and it's got a value in there*

*eh, does an add again expanded at the bottom*

*and it's adding the contents of an address to the contents of Y*

EX: What that is actually doing is adding the contents of address 50 plus Y

*AE: Let me go back*

EX: So it's the contents of, the address is 50+Y

*AE: Oh sorry yes, and it adds that to the accumulator. I made an assumption that, I was probably confused by other assemblers*

EX: It's not a straight forward addressing technique

*AE: Decrement Y yeh*

*branch*

*it's just round the loop again now*

*with Y decremented*

*there it's doing the addition, lets see if I can get it right, and so it's going to add ... em whatever is at Y and 51*

EX: Which is highlighted

*AE: Which is highlighted, and put it into the accumulator, which has got 7 in now, and it will now become, eh!*

*...adding the*

EX: It's probably already added it

*AE: Yes it has, it had 2 in it before*

*and then its done the addition which makes it 7 ok*

*storing it*

*decrements Y, which sets some bits in the condition register*

*right*

*ok that was nice*

*when it was doing the branch plus the appropriate bit in register is highlighted.*

EX: That happens all the way through

*AE: Yeh, makes it clear what its actually testing*

*going through it all again*

*3 and 6 is 9, good*

*storing it*

*decrements Y now Y is gone to zero*

*so the N bit is 1*

*so its going to do the test for the branch plus*

*and it doesn't branch*

*end of program*

EX: Talking about the machine code, with the mnemonics seeing them side by side. I don't know why you would want to see the machine code values, maybe it would be so the story is a truer story of whats going on

*AE: Yeh*

EX: Also on the first pass you could go through changing all the mnemonics to the machine code, and you change things like NEXT to machine code addresses. I think that complicates the issue.

*AE: I think so to, unless you trying to teach people machine code*

EX: Any comments about the different

*AE: Yeh I quite liked that because reading through the program just on it's own and with your em hints about it I had some idea what it was doing, but actually following it through on here was much clearer. It's perhaps a shame youve got limitations on the screen, but it's a shame you can't see all the relevant memory there, which presumably this is 9 7 6, but you can't see the 6. Em otherwise I hink its fine*

MS's Protocol for the Assembler APT-0

*MS: OK interaction, database. Step and start. That doesn't seem to have anything to do with the listing of the program at all, I'll go forward and see what happens.*

EX: That's actually what you would call

*MS: Oh I see yes. Instantiates values to mnemonics, are so this appears to be assembly time we are talking about, right the assembly code, forwatd backward stop, and there is a v against something, oh that's an arrow*

EX: That's explaining that there is more program below

*MS: I see, eh right number = 3 so*

*go forward, database contains number = 3, right so evidently this is something to do with compile time, I'm suspecting that its going to have compile time, execution time mixed up.*

*assigning values to mnemonic right*

*so it goes through them*

*all this seems to be doing is moving things into the database, this is into the compilers database, the assemblers database rather eh*

*assigning value to mnemonic*

*part 1 no change*

*no change in what*

*Ah right in fact its t*

*his is what I would call the first pass of the assembler, I've just caught on. em pointer 1 has a value so its replaced the mnemonic by the value.*

*the same applies*

*right*

*and evaluates to 11*

*it's doing the evaluations at the time*

*I think I completely understand whats going on here its.. its equivalent to the first pass of the assembly. I'm not sure whats going to happen whaen it comes to labels. thats going to be interesting I think*

*at the moment its only dealing with defined constants*

*so that seems straightforward*

*here comes a label*

*and er yes the hash is left out as I would have expected cause that's part of the instruction really, number has got a value, yes so it's done nothing special with*

*the label at all although in fact I would have expected something to happen there because the blockadd is, the information from the blockadd I would expect to see that in he database. The address of .. no maybe not*

*maybe that only comes when its assigned addresses to all the code*

*so right we're going through all these defined constants,*

*second part execution of the code . No I wouldn't agree that that comes next I would have thought theres assigning*

*removing assignment mnemonics I wonder what that means right*

*that's obvious what that means, right so in fact we are moving into execution now I don't think thats right because, I think another part of the assembley process should have been to replace the labels blockadd and next, well should have assigned addresses to all the code, and replace the labels with those addresses, thats the parallel process to replacing the symbolic constants that I've just seen, anyway*

*move to first instruction , evidently this is the time things are happening. I can't remember what an SED instruction is*

EX: It sets it to decimal

*MS: Ah right. execution that has an affect on the internal state, oh yes. Has an affect on th decimal flag on the whatever its called. Fine.*

*Load 3, I expect to see a constant of 3 because load immediate instruction, see a constant of 3 appear in the accumulator and indeed it has.*

*STA in address 10. I immediatly think where's memory.*

*we shall see no doubt, show address and contents*

*here it is, memory has materialised half way down the database and there is address 10 and I'm expecting to see that 3*

*appearing in there, and it has done right*

*load A 2*

*and off we go*

*store it in address 11*

*a sequence of load and stores*

*I can't get very excited about store in address 50, oh and we see another handy bit of the database, and a 6 has appeared in there*

*oh I see the edge of the memory sought of bulges to accomodate the number*

*actually I seem to remember Tim making this point, thats not very clever because it implies that memory can have no contents, whereas in fact it can't there is bound to be something in there, random rubbish or something*

*having said that possibly it could be marked so it would show that we don't know whats in it.*

*but I think it would be more realistic if it where random rubbish*

*right now we come onto some more interesting instructions here, load y with 2, suddenly some flags have changed, wasn't really expecting becuase I wasn't thinking about flags, but yes the negative flag and the zero flag where both clear*

*clear the carry flag, oh no haven't finished doing that yet*

*execution right*

*now next instruction clear carry*

*er execution, and the carry flag has indeed been cleared*

*load A with 10 indexed by Y, we can see Y which we loaded up there which I forgot to notice, which I expect to see getting loaded with the one from address 12*

*and its telling me where it's going to come from, yes good, nice*

*low and behold there it happens*

*er add with carry 50 indexed by 2*

*so I expect to see that one light up and it does*

*and we've got 1 and 4, store it in a 100 indexed by y*

*A new bit of database is going to appear in that corner I suspect*

*and ah, now for some mysterious reason on this bottom line, which it hasn't shown me previously before, it's given me a hint of what the known value of y which I can see up there is*

EX: It was actually there before

*MS: Oh was it I just didn't notice it, oh right. Lets go back and have a look*

*oh yes it was there all along and I never even noticed it, cause I was cleverly saying what I thaught it was going to do*

*store in a 100,Y, corresponds to 102*

*er I don't think thats entirely true, yes I suppose it is*

*show addresses and contents, yes there they are with the mysterious blank contents*

*and there's the answer 5 going in there*

*decrement Y*

*there it goes, it was 2 and now its 1*

*branch on plus to next, and that is possibly going to show up the negative flag*

*Yes indeed it does. Yes I don't like this whole aspect of the thing its as if the processor when executing it new what 'next' was although it doesnt. The processor no more knows what next is than it knows whatever the mnemonic was that was in there, pointer y. Both the pointer y and te next would be removed at assembly time so in one sense, yeh I felt they should have been dealt with consistently*

*Its clear whats happening, I dont think the thing is unclear*

*but it's not telling a consistent story in that sense*

*yes clear carry execution, yes*

*same sought of thing goes on*

*hang on why did address 10 and 11 light up there*

*thats interesting*

EX: It's scrolled

*MS: Load A, ah right 10l,Y and Y was 1, I thought I saw, yes I did see it move down*

*ah right now that was, I found that suprising, cause on seeing the grey band move down, I thought that it was the notion of the band moving was to draw my attention to something different, and I found it confusing there to see the band moving, and its really the scenery thats going past*

*never mind, load A with that*

*and add it*

*and so it goes round the loop again*

*and Y is no longer positive, thats interesting must have gone round twice already, em*

*presumably I have done*

*it lights up next although in fact its going to be irrelevant. Wait a minute Y has already been decremented to zero. The negative flag is still 0, it is going to do it 3 times, thats right. Yes I though it was going to not. Which shows in fact that seeing those two instructions together I was em carelessly assuming that the thing to look at was the Y register which your not, and on it having become zero I thought there's no more chance to do it, as if that was as it would be in some machine codes er a branch not on a flag but on the contents of a particular register, beeing zero. I think its more usual to be on a flag.*

*so on we go and do the third one*

*and we get the scrolling again on down the screen*

*branch plus the next*

*and it didn't, ended trace finished, ah right*

EX: So the main problem is that you say that something should happen in parallel with the first pass

*MS: Yes in fact its worse than that, because the real story that I would like to see it telling consists of 3 passes, in which the second on this appears not to have too much happening in it, em but*

EX: It would actually happen in 3 passes rather than 2

*MS: Oh yes becausse the code would, assuming that this was actually in the source file, it would go through the assembler and produce some object code, and than get executed thats two, but the assembler itself normally consists of two passes. In the process of producing the object code, is a two part process. Apart from the third part of executing it.*

EX: Was the first part alright

*MS: The first bit, turning those numbers into constants was alright as far as it went, that would be part of the first pass, but that should also include. You had an initial process of doing the equates didn't you, yes I'm not sure about the order but analogous to that it should have been putting into what you call the database here the value of blockadd, in fact it woulds be quite hard to do because you've done the whole thing without reference to what addresses these instructions are stored at*

EX: So the first pass would be putting an instruction at an address

*MS: It would be assigning an address to it. There would be a default, each instruction would be 2 or 3 bytes long, it would vary but these are 2 or 3 bytes long. It doesn't matter in this case how long they are but in others it would be important, and there is a default starting address at which the program would be loaded. It might just as well be zero, er and then the first part of the assembler should apart from handling the equates, should also assign addresses to each instruction which would then mean that by the time it came to this one it would know that it was at address 51 or whatever and that would, well blockadd doesn't really matter because that label isn't refered to anyway, supposing it where next and next where 51 then that should assign 51 to next exactly parallel to how you assigned 3 to NBR in the thing on the right hand side there. Thats the first part. The second part was pretty similar er but during the second pass it would use the value of 51 or whatever in place of that next, so it would actually turn this into branch plus 51*

EX: That's what the second pass does, replace..

*MS: Well yes that would come under another point which I hadn't thought of, in a way its possibly slightly confusing to say that anything is replacing anything. What the assembler is doing is producing object code, in the object code isn't LDA, there is a bit pattern which corresponds to the actual machine instruction and at the same time it's not, any of the labels haven't got the name of the label or anything to do with it. It's just a number or bit map or whatever. Apart from labels and the 2 passes and everything you could argue that the presentation is slighly odd in that the assembler doesn't make a particular seperate exercise of changing any occurances of PTR1 in the program, changing as it where changing PTR1 to be 10, er. That presentation seems to imply that as a special exercise and apart from anything else it goes round changing things, and at some point there is a program that looks like that, like it is on the screen with the instructions and the hashes, and there never is really because the actual replacing, the use of the value PTR1 happens at the same time as the other processes of*

*changing the mnemonic into an op code happen. Processes that it would not be terribly helpful to show on this display.*

EX: So you would change the PTR1 into the address of 10

*MS: It would just change it into a 10 in fact, just as you showed it do. The point is it wouldn't do it in isolation at the same time as that processis happening, which is perfectly realistic that you show, of replacing PTR1 with 10 it would also replace STA with the code and it would throw out all the spaces. Imean it's not really operating in place at all. What it's doing is to produce a new file, or section of memory, containing the replacment for STA and the 10 for PTR1, there is no place where the STA belongs alongside the 10 really*

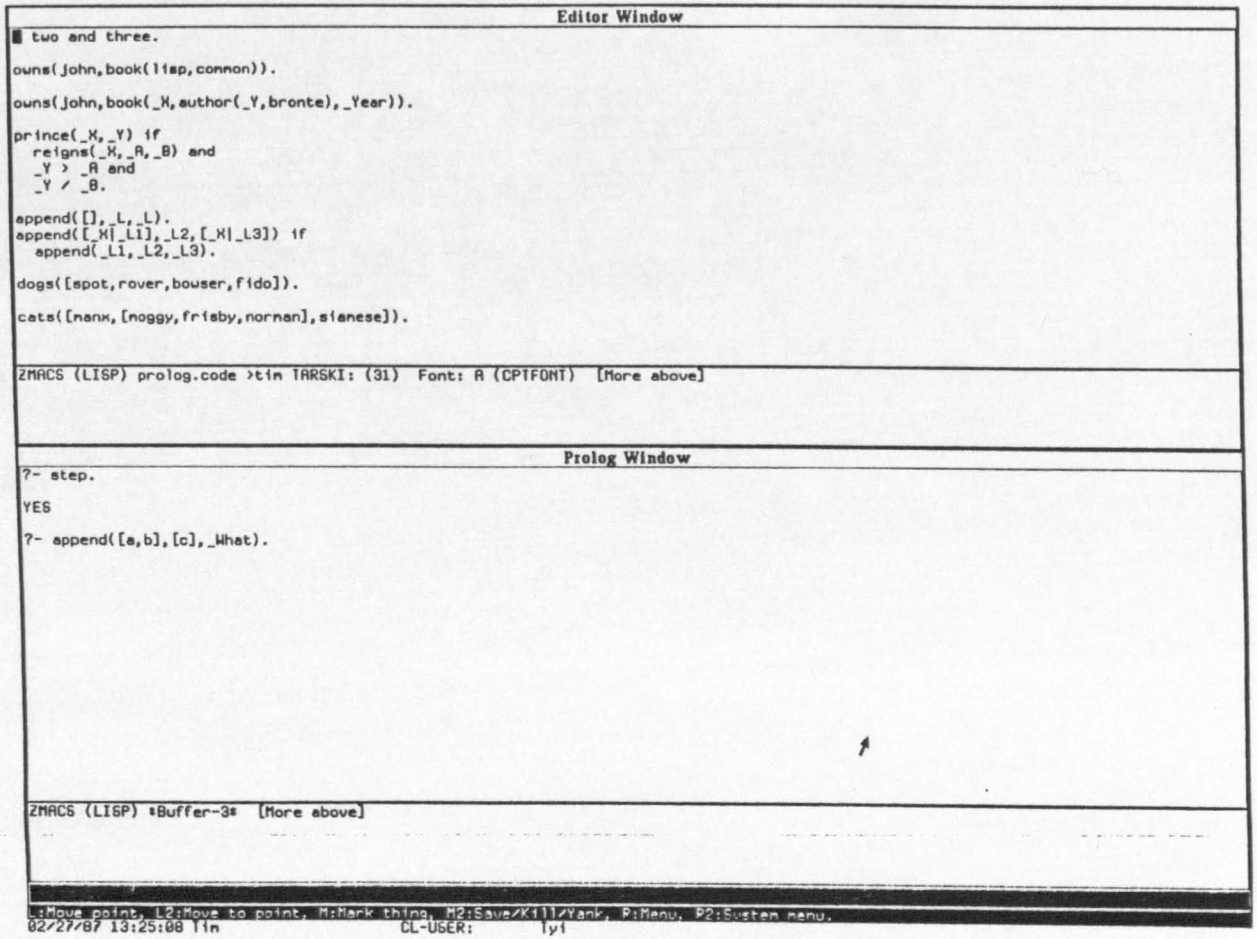
EX: So the problem is that if you try to show the real story what you would get on screen would be jibberish really

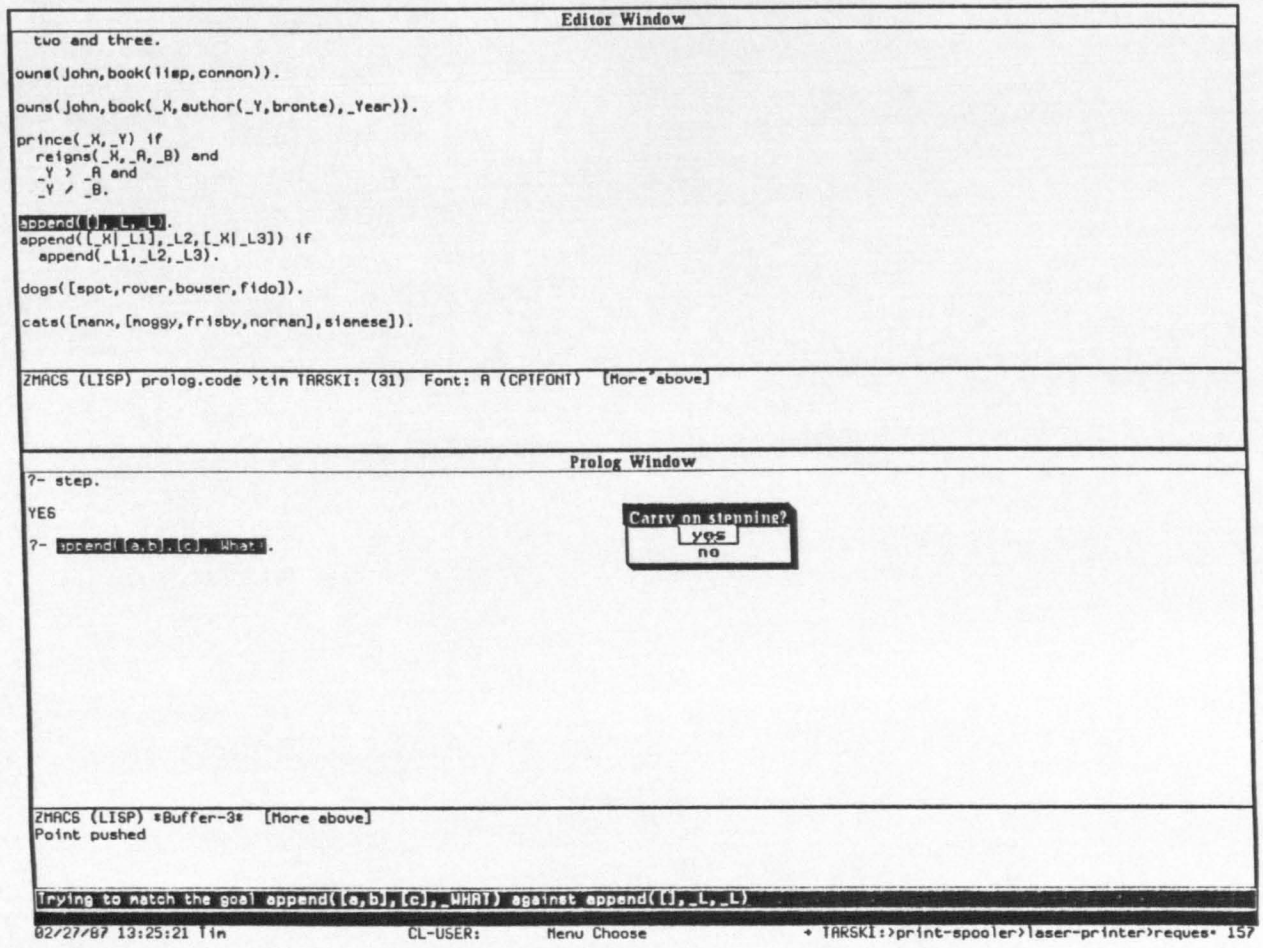
*MS: Possibly. All the details that I am talking about are not in some sense relevant, there are some times you wouldn't need to know them and other times you would, but it's a little bit odd to see certain things going on er to those changing in place although they do, no no no, I mean they don't at all. That is one version of the story you can give, you would give that if that was your particular, if that is what the student needed to know at the time. Its not a lie really in the sense that everything is a lie, but it is an incomplete story. I think it would be more typical on a machine code trace of this sought to have em something like a full assembler listing. What you get on an assembler listing is not only te mnemonics that you have written but a translation of them into hex code, and if you did that. I don't know if you could get over the difficulties of fitting it on the screen, but suppose you did, and it was not too much of a distraction anyway. What you should probably represent in the assembly process, is that you would leave the source code unchanged. That would still say PTR1 but then as part of the assembly process that you are depicting the machine code for STA would appear in another column. Thats the sort of more accurate way, whether its more helpful, because its just filling up the screen with what people don't want to know, I don't know.*

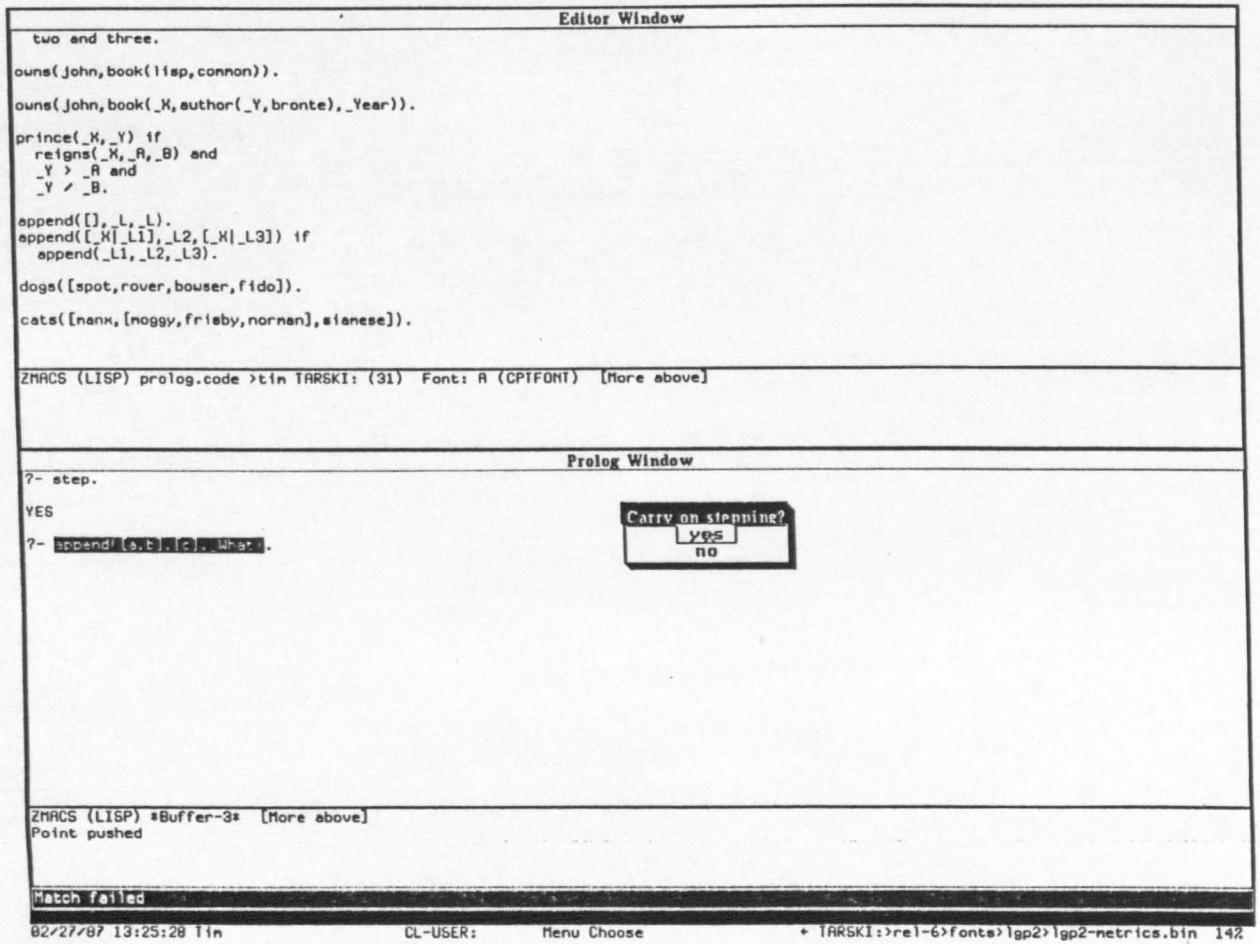
## APPENDIX-K

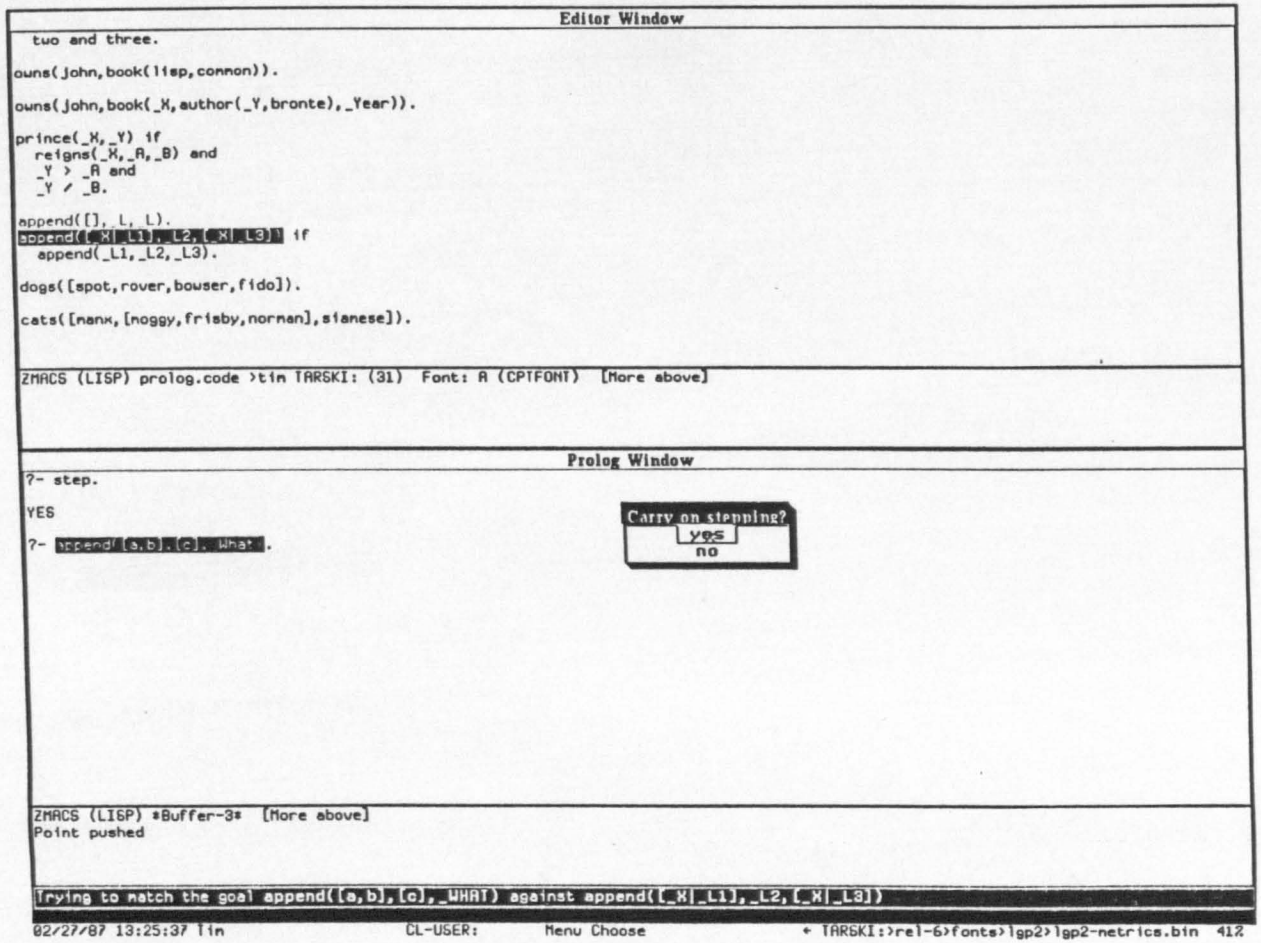
Appendix K presents a complete APT listing of the execution of a Prolog program demonstrating list manipulation. The listing consists of a series of screen snapshots which have been taken at each step of the APT display. The program used in this example is 'append' which is defined as follows.

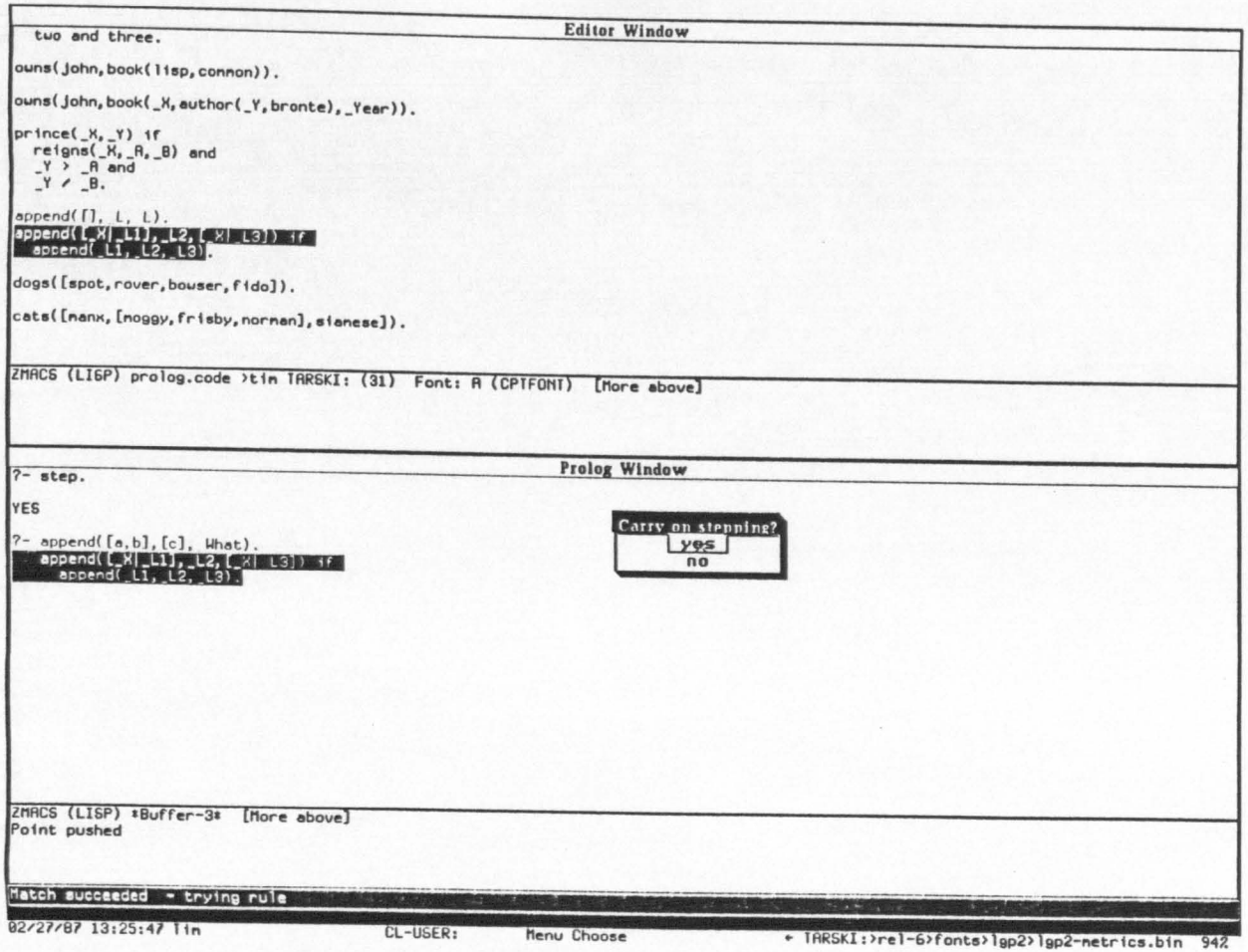
```
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3] if
  append(_L1,_L2,_L3).
```

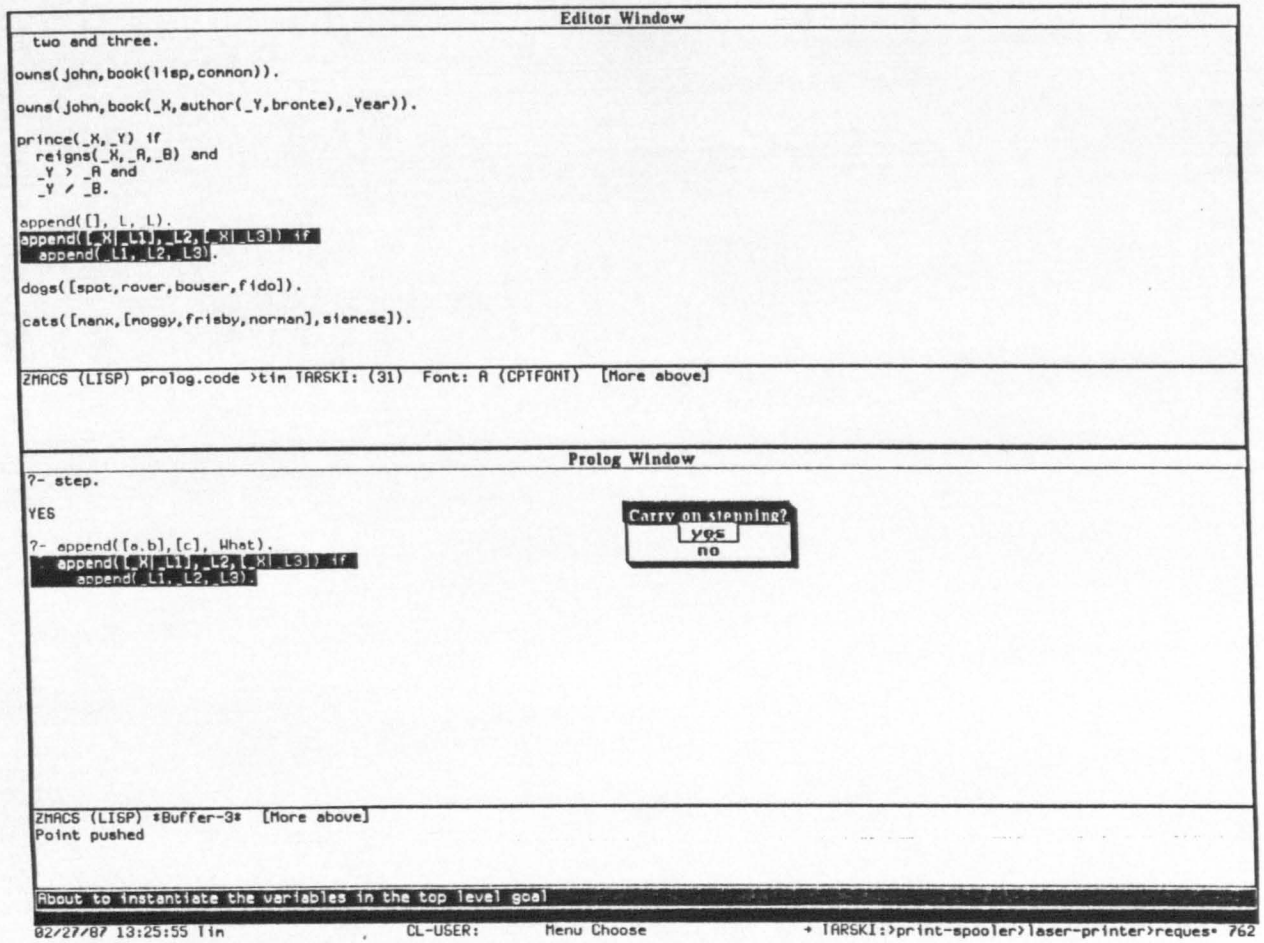

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
ZMACS (LISP) *Buffer-3* [More above]
:Move point, L2:Move to point, M:Mark thing, M2:Save/Kill/Yank, P:Menu, P2:System menu.
02/27/87 13:25:08 Tim CL-USER: (ty)

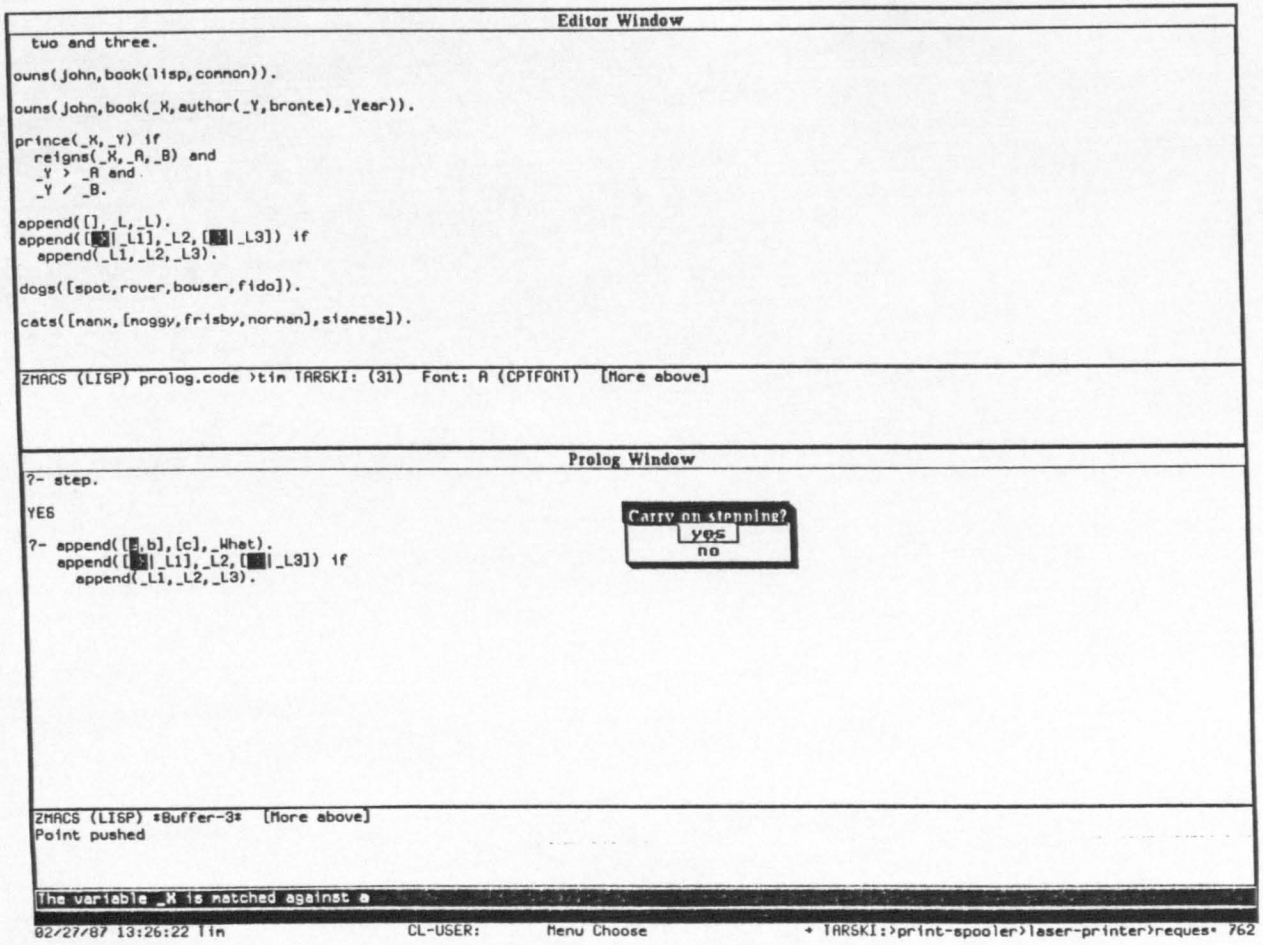

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],What).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal append([a,b],[c],_WHAT) against append([],_L,_L)
02/27/87 13:25:21 Tim
CL-USER:
Menu Choose
TARSKI:>print-spooler>laser-printer>reques 157

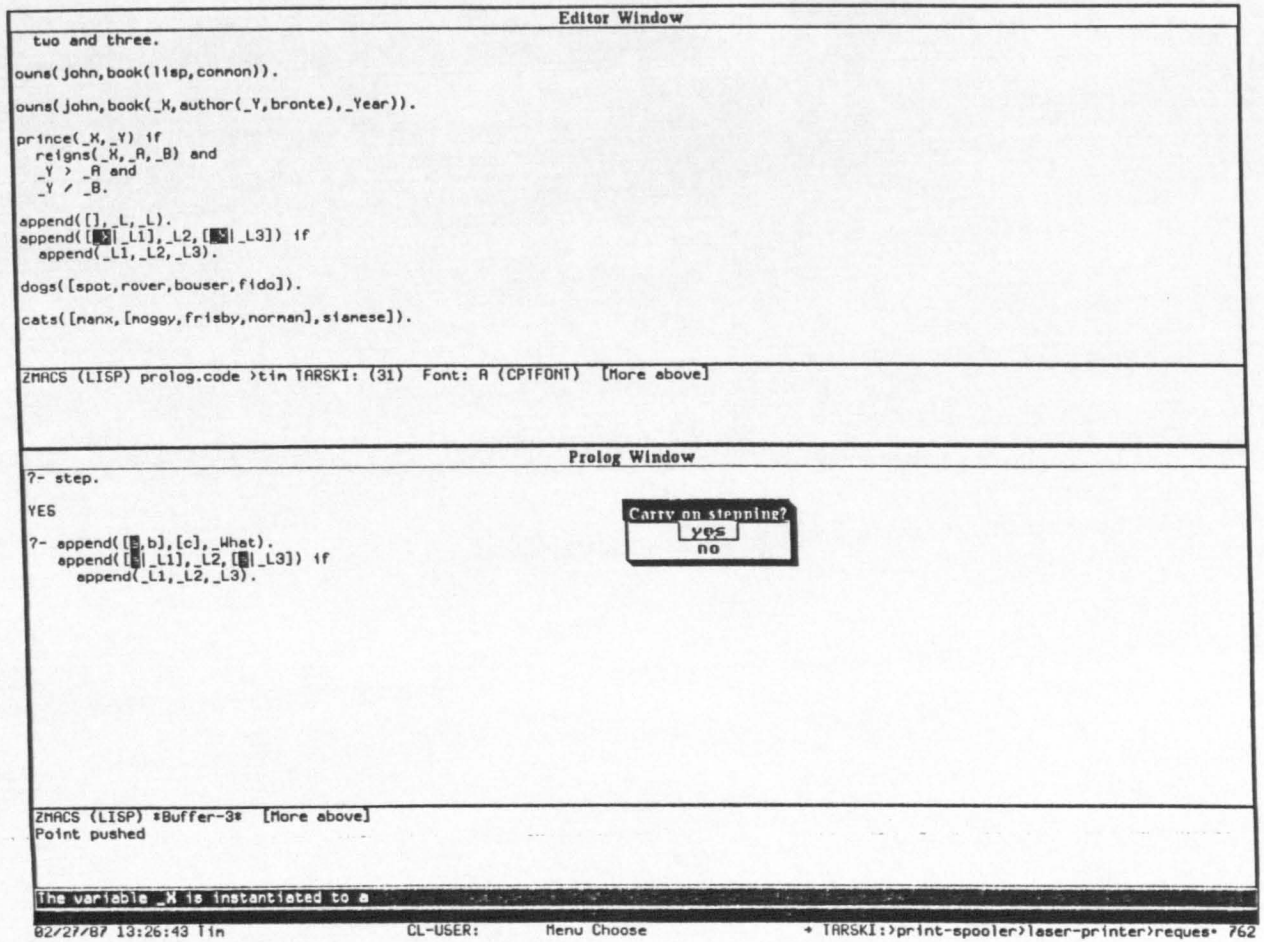

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[moggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append(a,b,[c],What).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed
02/27/87 13:25:28 Tim
CL-USER:
Menu Choose
TARSKI:>rel-6>fonts>lgp2>lgp2-metrics.bin 142

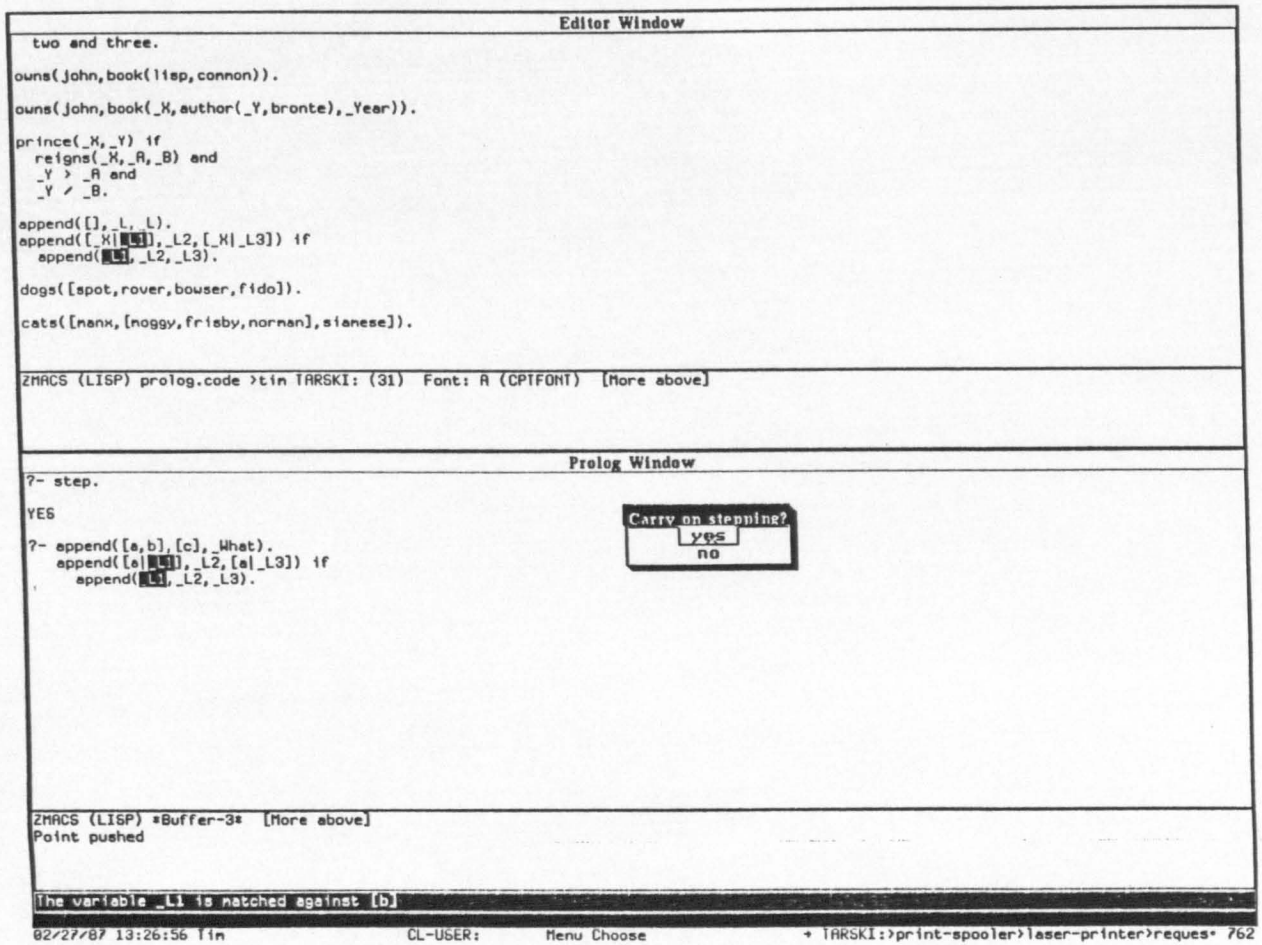

Editor Window
two and three.
owns(john,book(lisp,conmon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([], L, L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],What).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal append([a,b],[c],_WHAT) against append([_X|_L1],_L2,[_X|_L3])
02/27/87 13:25:37 Tim
CL-USER:
Menu Choose
TARSKI:>rel-6>fonts>lgp2>lgp2-metrics.bin 41%

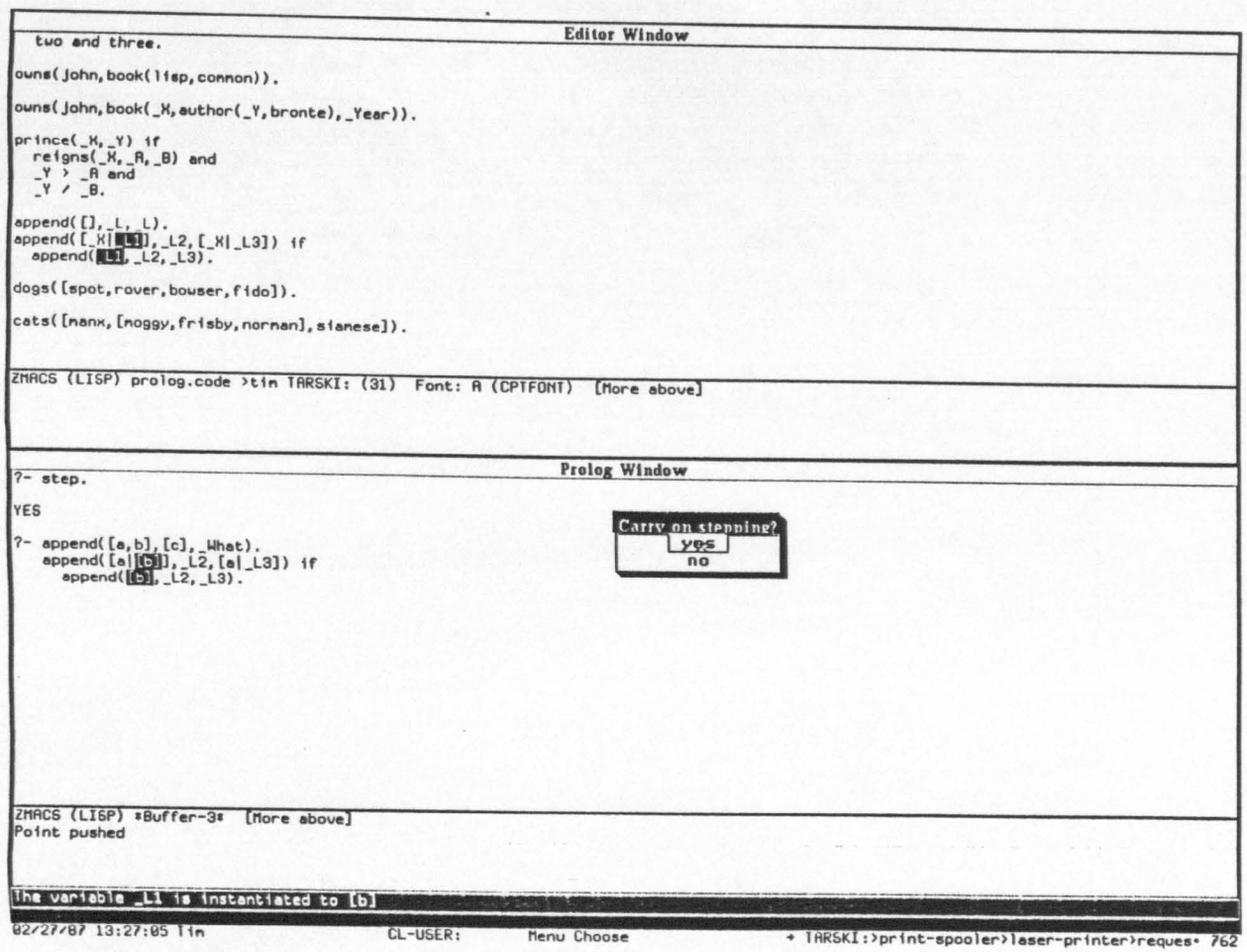

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([], L, L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c], What).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded - trying rule
02/27/87 13:25:47 Tim
CL-USER:
Menu Choose
TARSKI:>rel-6>fonts>lgp2>lgp2-metrics.bin 942

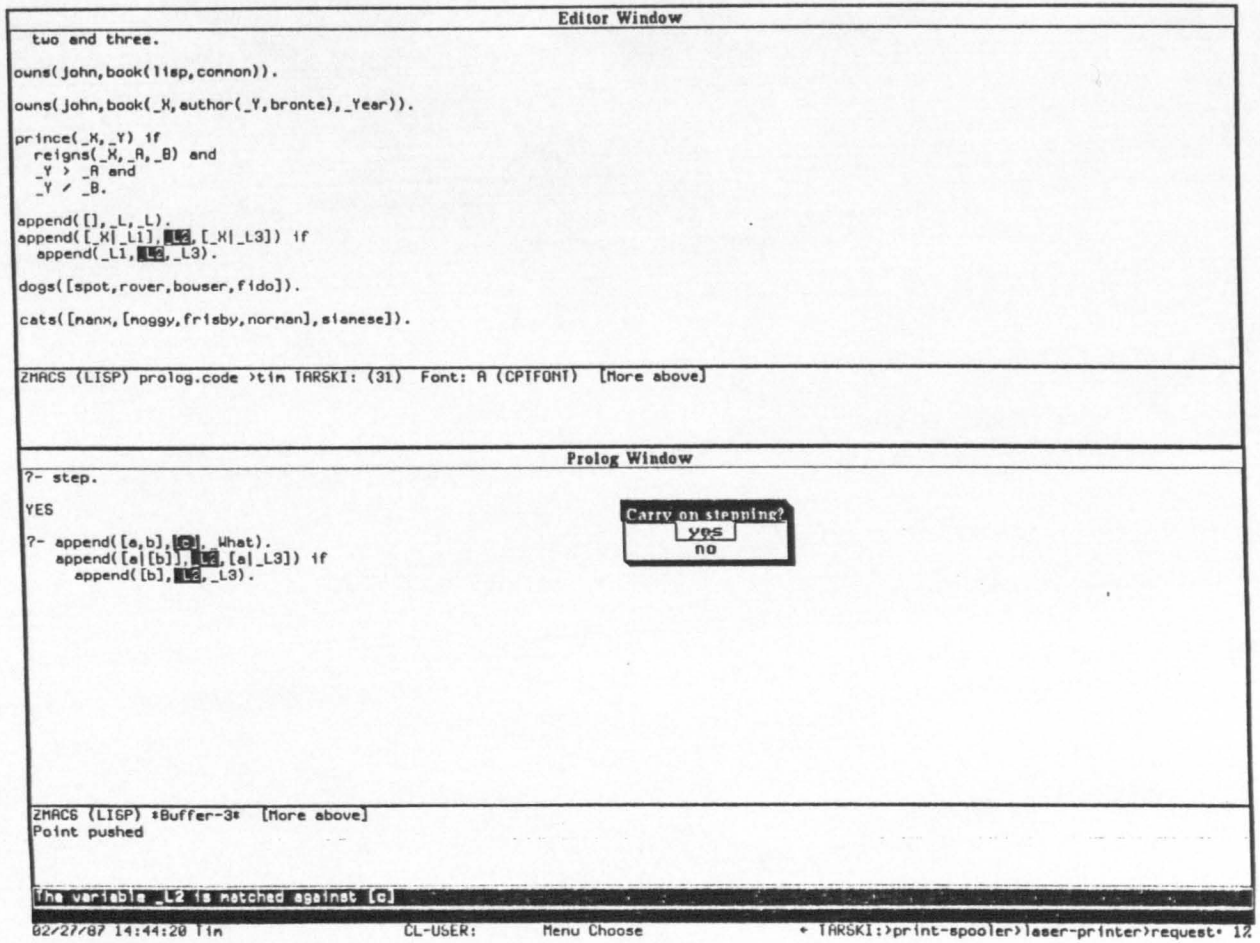

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([], L, L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1, _L2, _L3).
dogs([spot,rover,bouser,fido]).
cats([nanx,[noggy,frisby,nornan],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c], What).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
About to instantiate the variables in the top level goal
02/27/87 13:25:55 Tim
CL-USER:
Menu Choose
+ TARSKI:>print-spooler>laser-printer>reques* 762

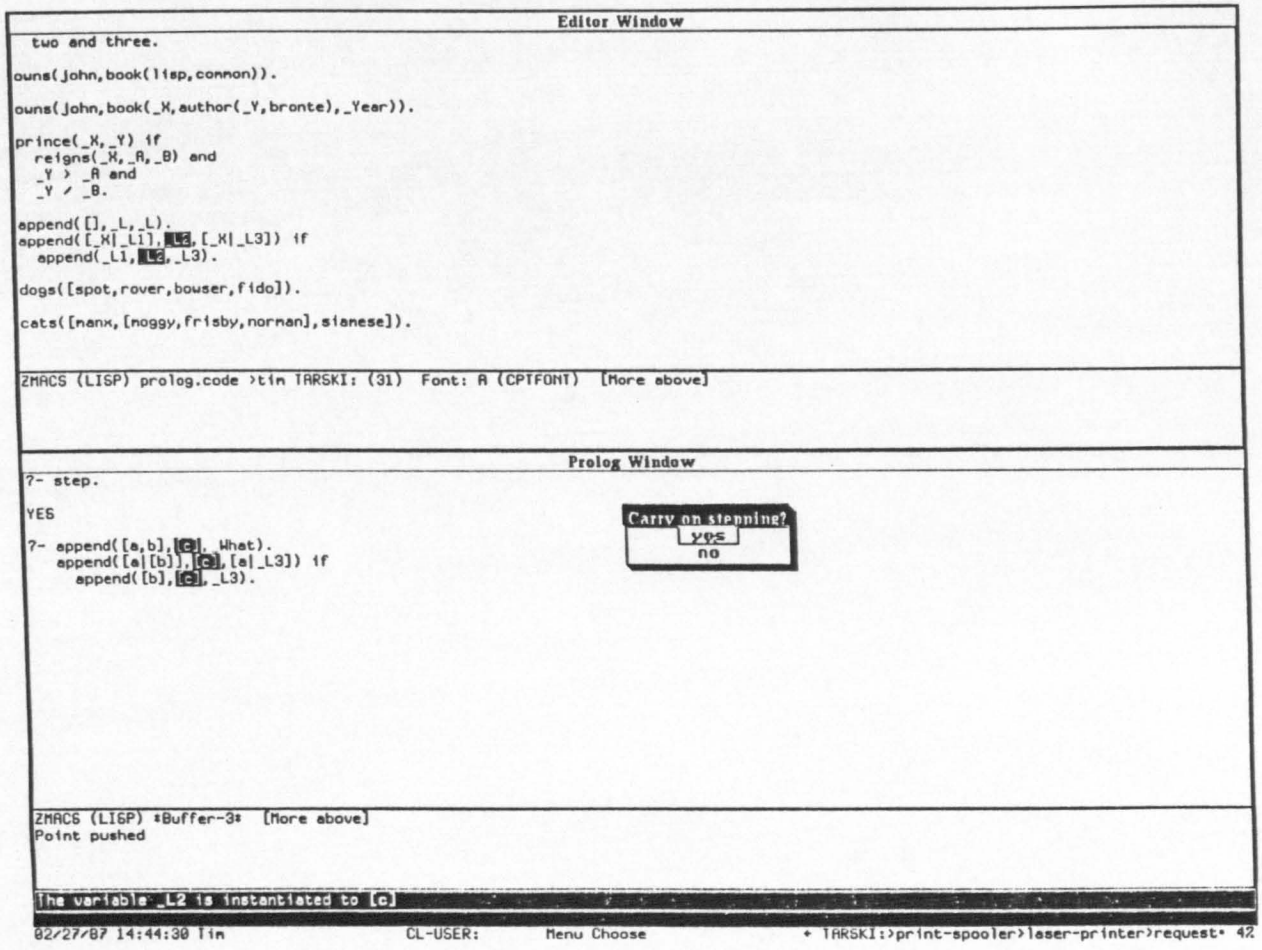

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_|_L1],_L2,[_|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([_|_L1],_L2,[_|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _X is matched against a
02/27/87 13:26:22 Tim
CL-USER:
Menu Choose
+ TARSKI:>print-spooler>laser-printer>reques* 762

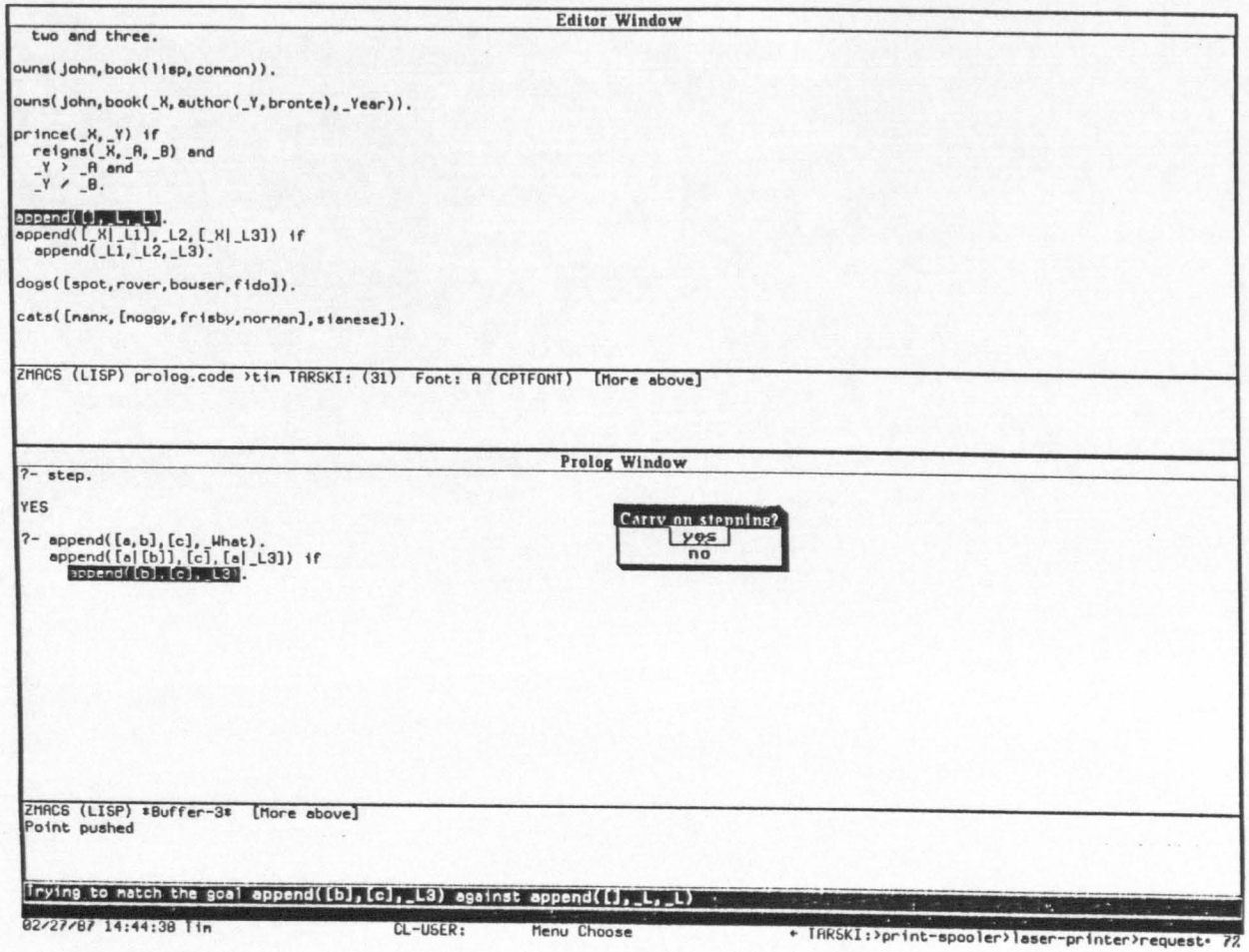

Editor Window
two and three.
ouns(john,book(lisp,connon)).
ouns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|_L1],_L2,[a|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _X is instantiated to a
02/27/87 13:26:43 Tim
CL-USER:
Menu Choose
+ TARSKI:>print-spooler>laser-printer>reques+ 762

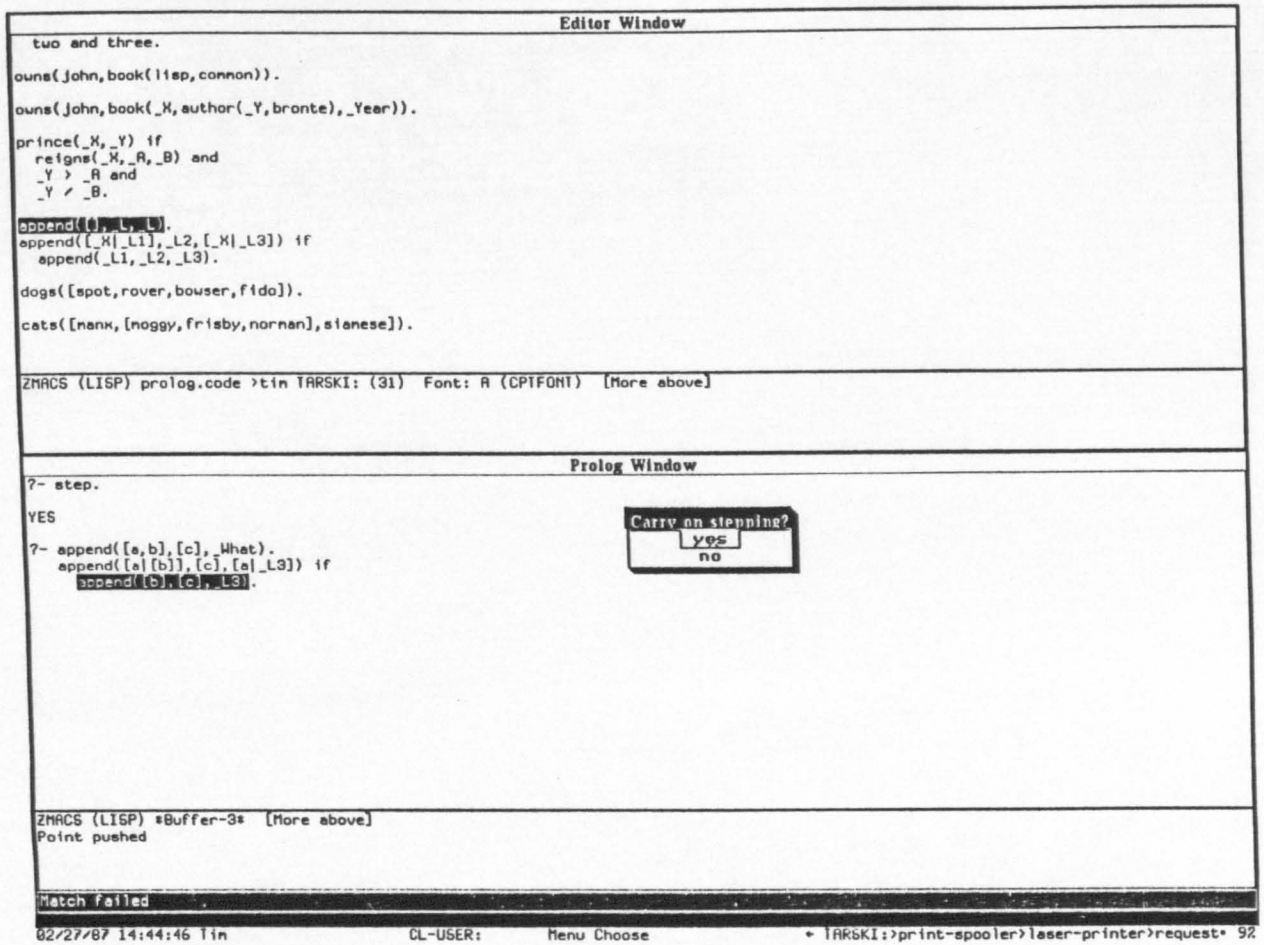

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
Carry on stepping?
yes
no
?- append([a,b],[c],_What).
append([a|_L1],_L2,[a|_L3]) if
append(_L1,_L2,_L3).
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L1 is matched against [b]
02/27/87 13:26:56 Tim
CL-USER:
Menu Choose
+ TARSKI:>print-spooler>laser-printer>reques* 762

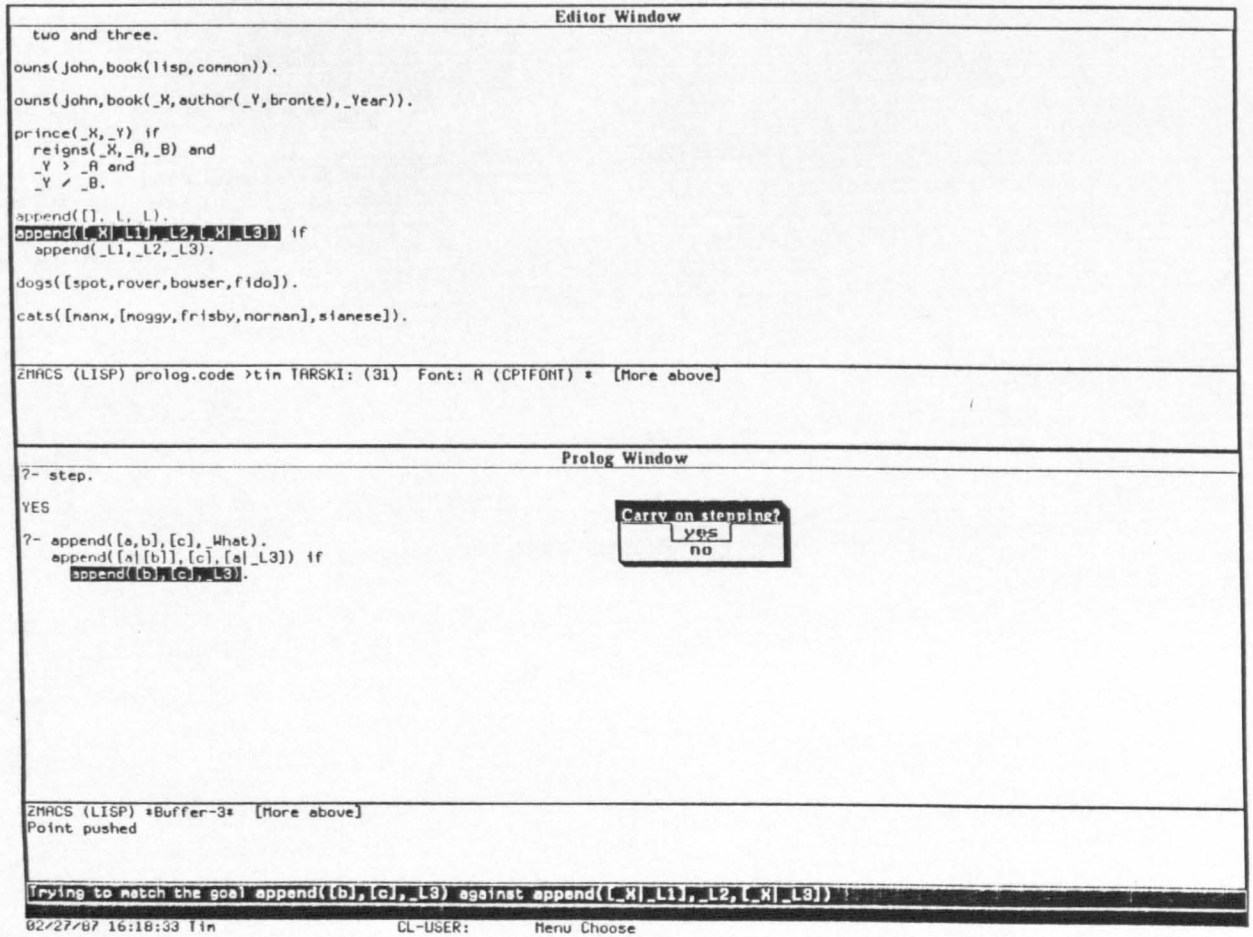

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],_L2,[a|_L3]) if
append([b],_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L1 is instantiated to [b]
02/27/87 13:27:05 Tim
CL-USER:
Menu Choose
TARSKI:>print-spooler>laser-printer>reques* 762

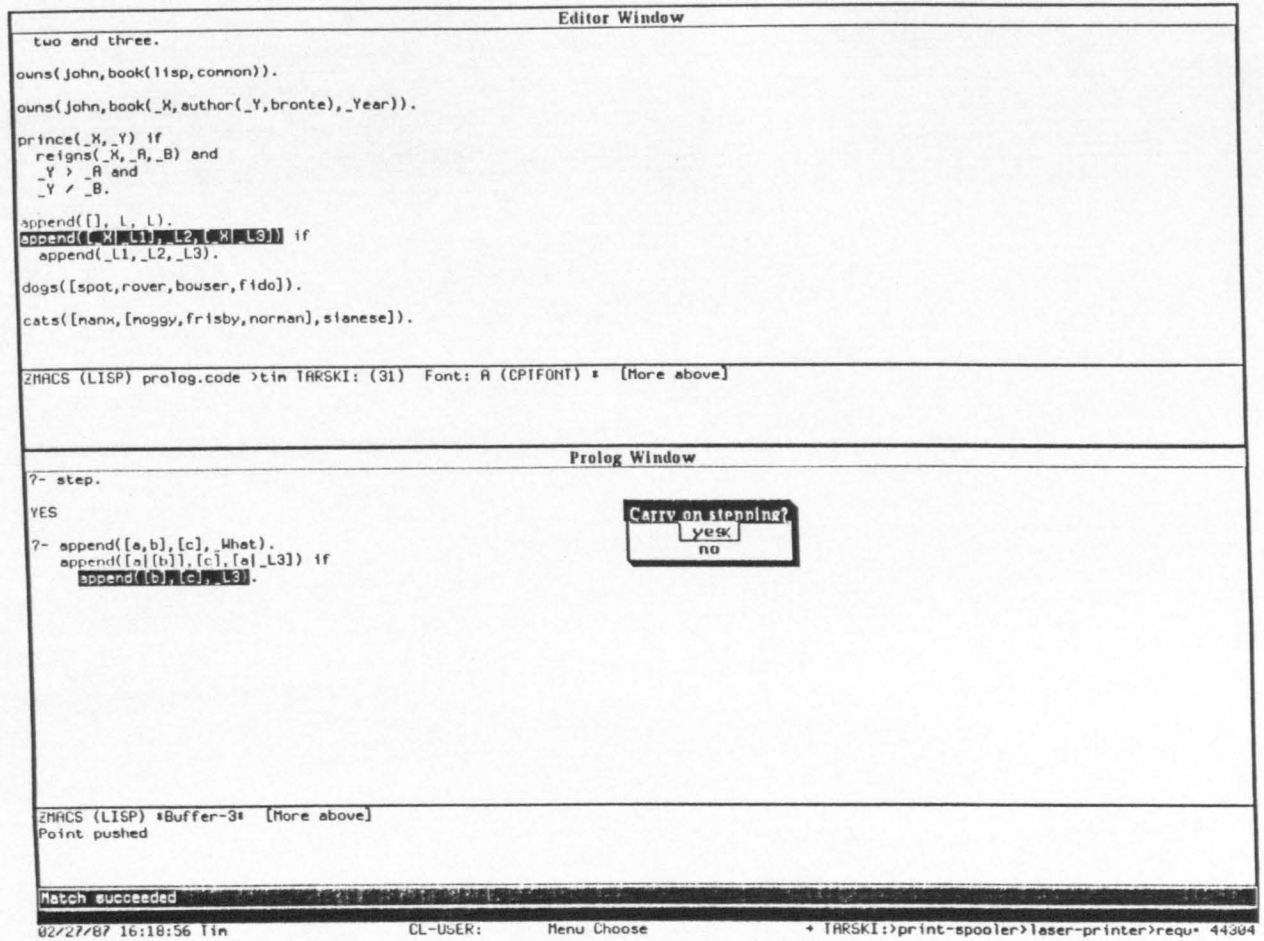

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],_L2,[a|_L3]) if
append([b],_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L2 is matched against [c]
02/27/87 14:44:20 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>request* 12

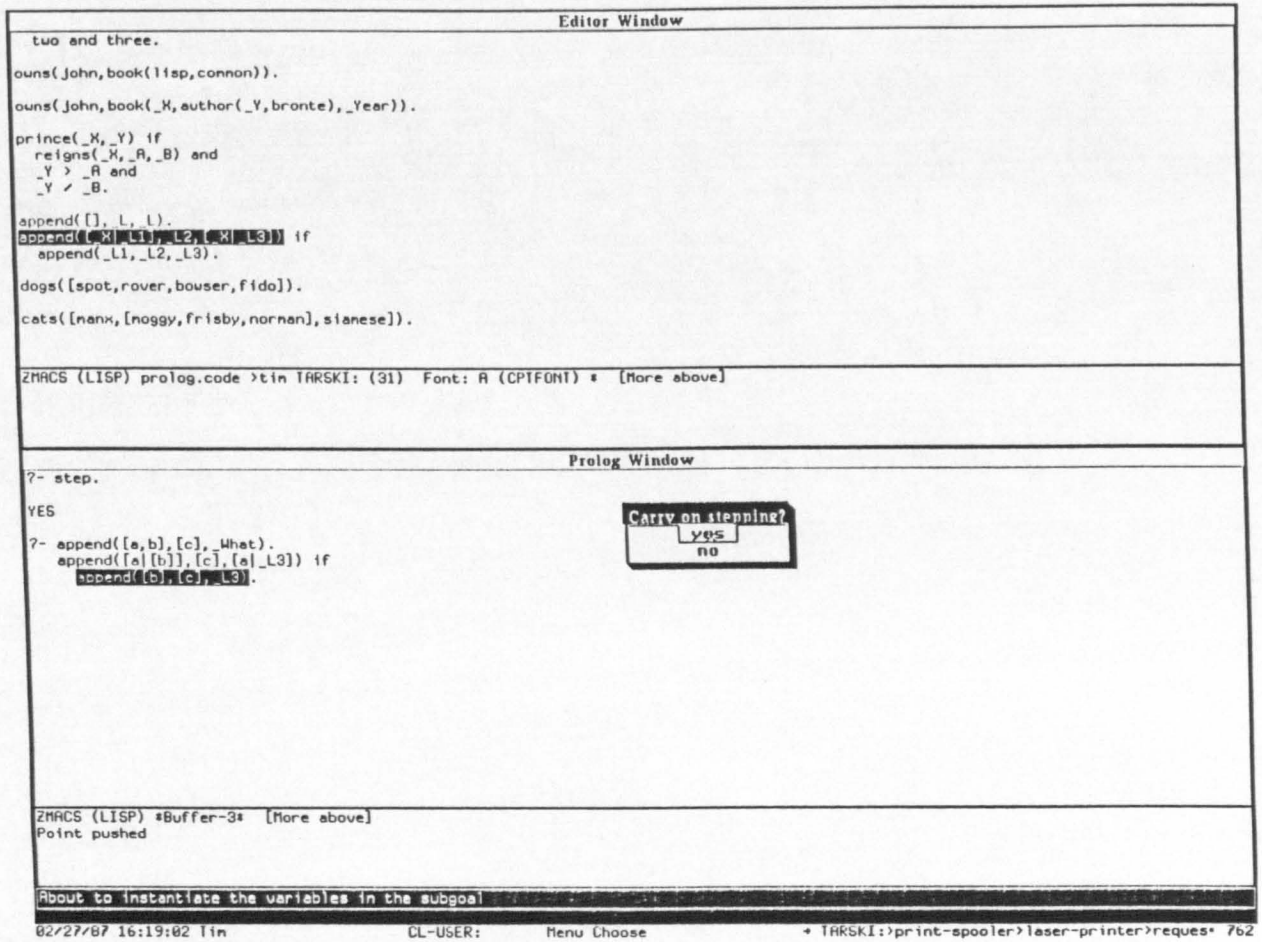

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],L2,[_X|_L3]) if
append(_L1,L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_L3]) if
append([b],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L2 is instantiated to [c]
02/27/87 14:44:30 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>request* 42

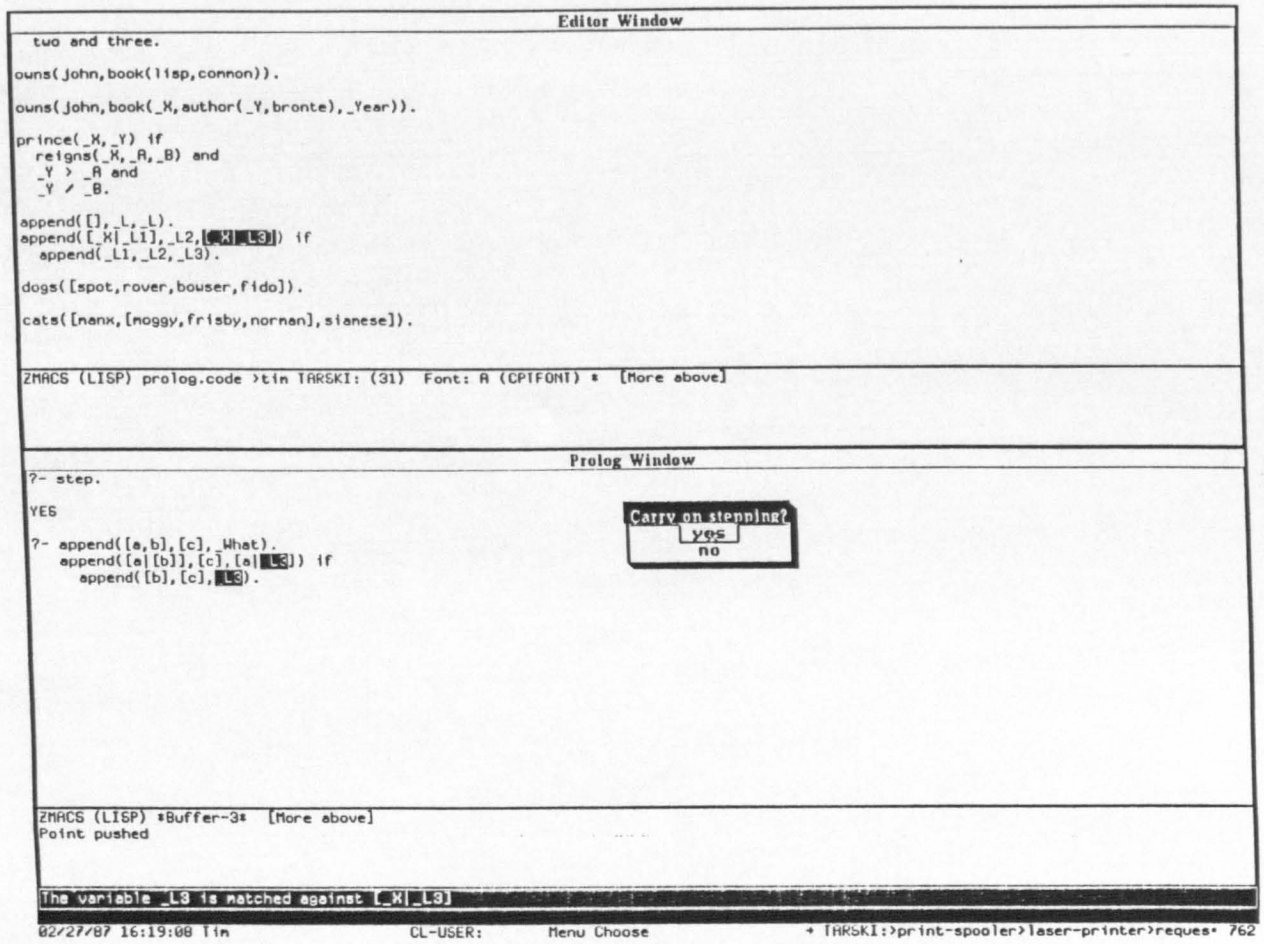

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_L3]) if
append([b],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal append([b],[c],_L3) against append([],_L,_L)
02/27/87 14:44:30 Tim
CL-USER:
Menu Choose
TARSKI:>print-spooler>laser-printer>request* 72

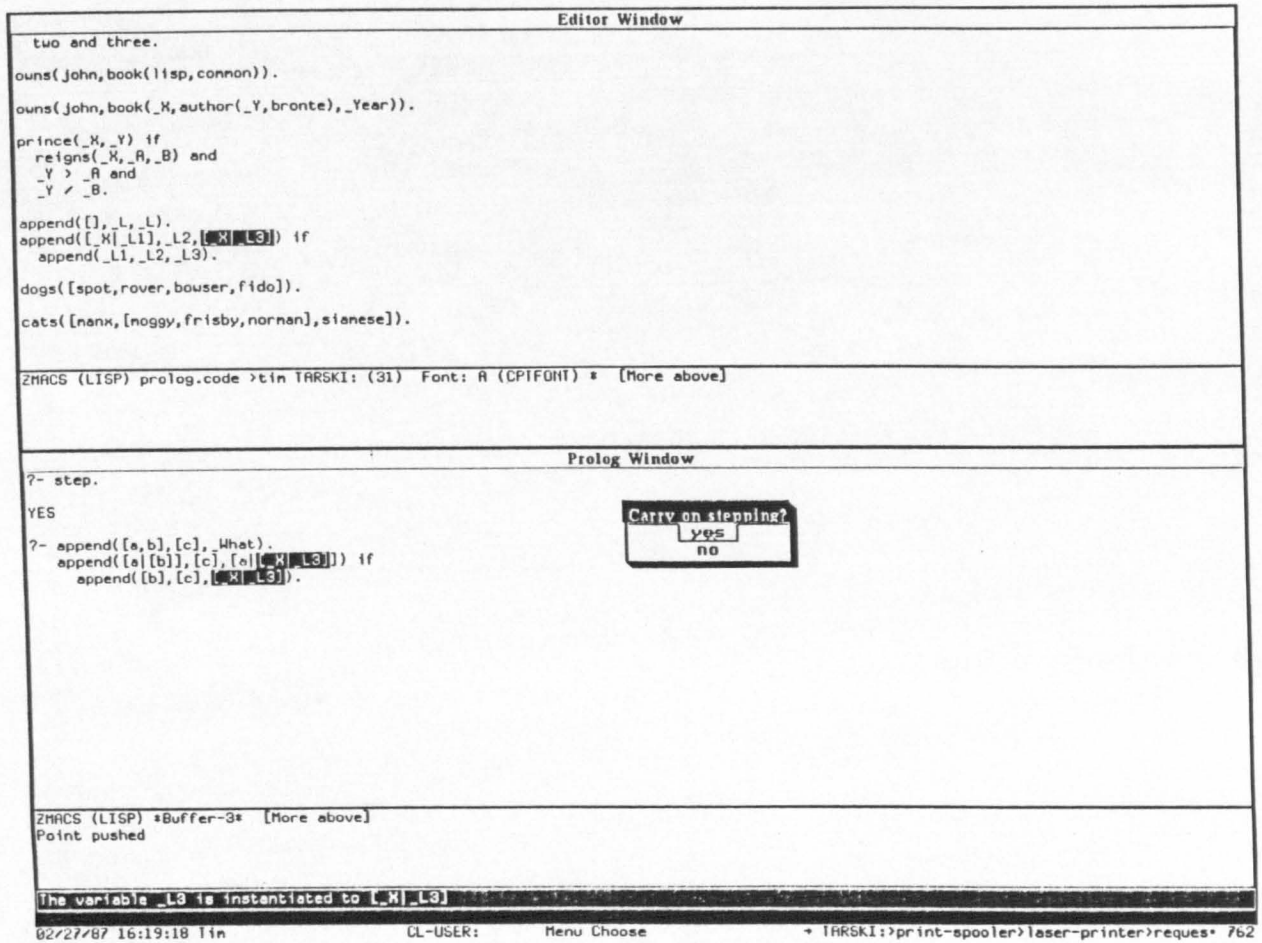

Editor Window
two and three.
owns(john,book(lisp,conmon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_L3]) if
append([b],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match failed
02/27/87 14:44:46 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>request* 92

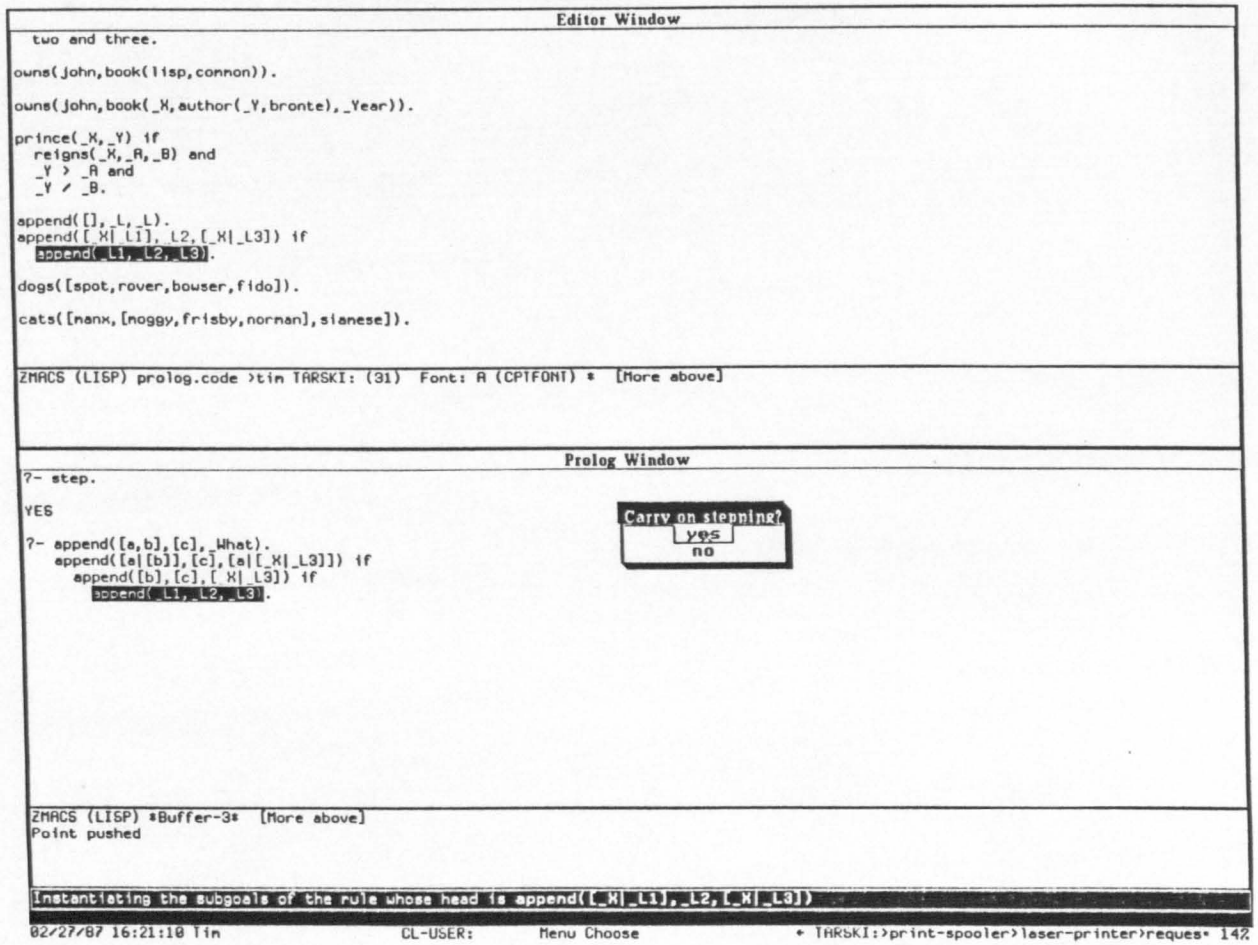

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([], L, L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_L3]) if
append([b],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal append([b],[c],_L3) against append([_X|_L1],_L2,[_X|_L3])
02/27/87 16:18:33 Tin
CL-USER:
Menu Choose

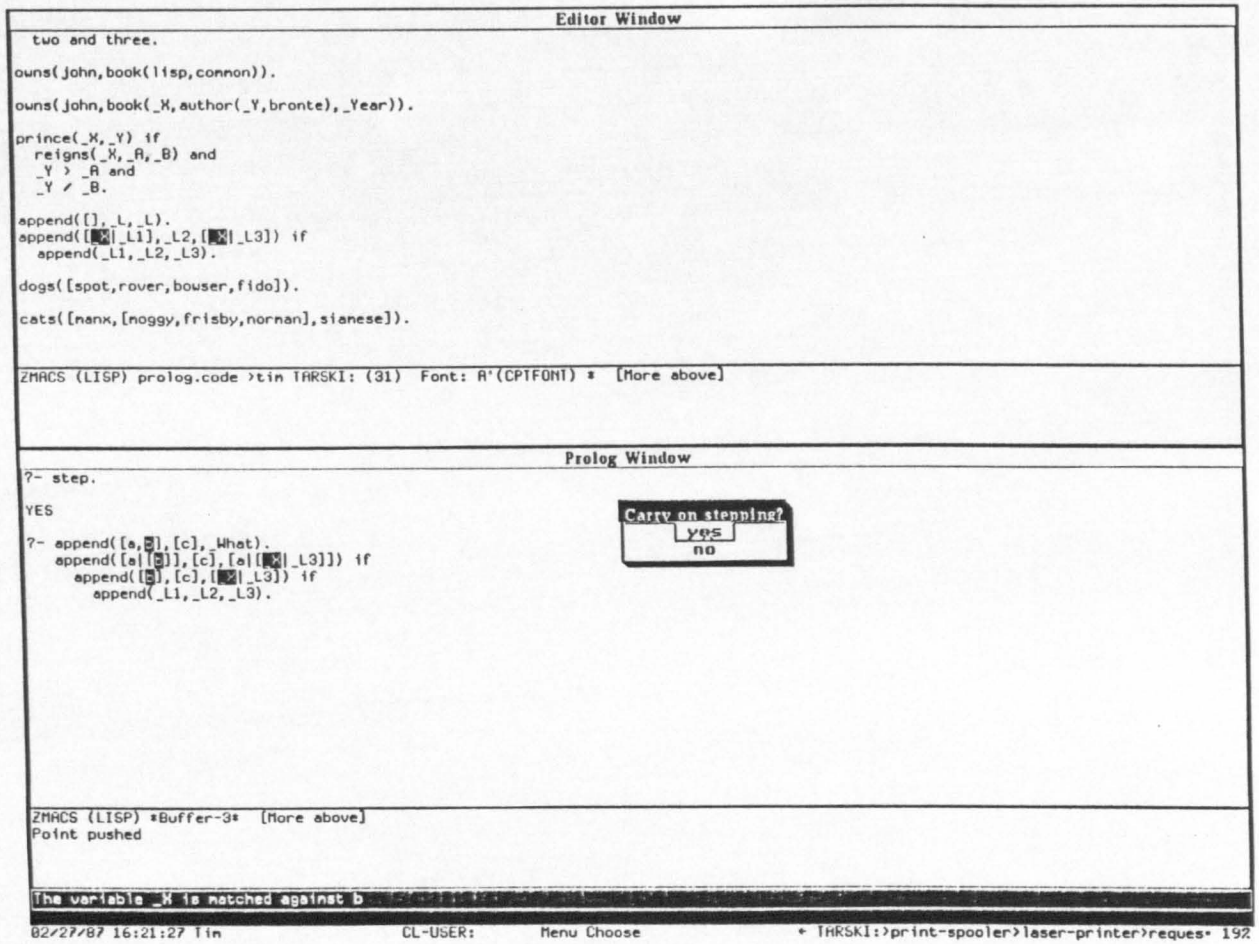

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([], L, L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_L3]) if
append([b],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded
02/27/87 16:18:56 Tim
CL-USER:
Menu Choose
+ TARSKI:>print-spooler>laser-printer>requ* 44304

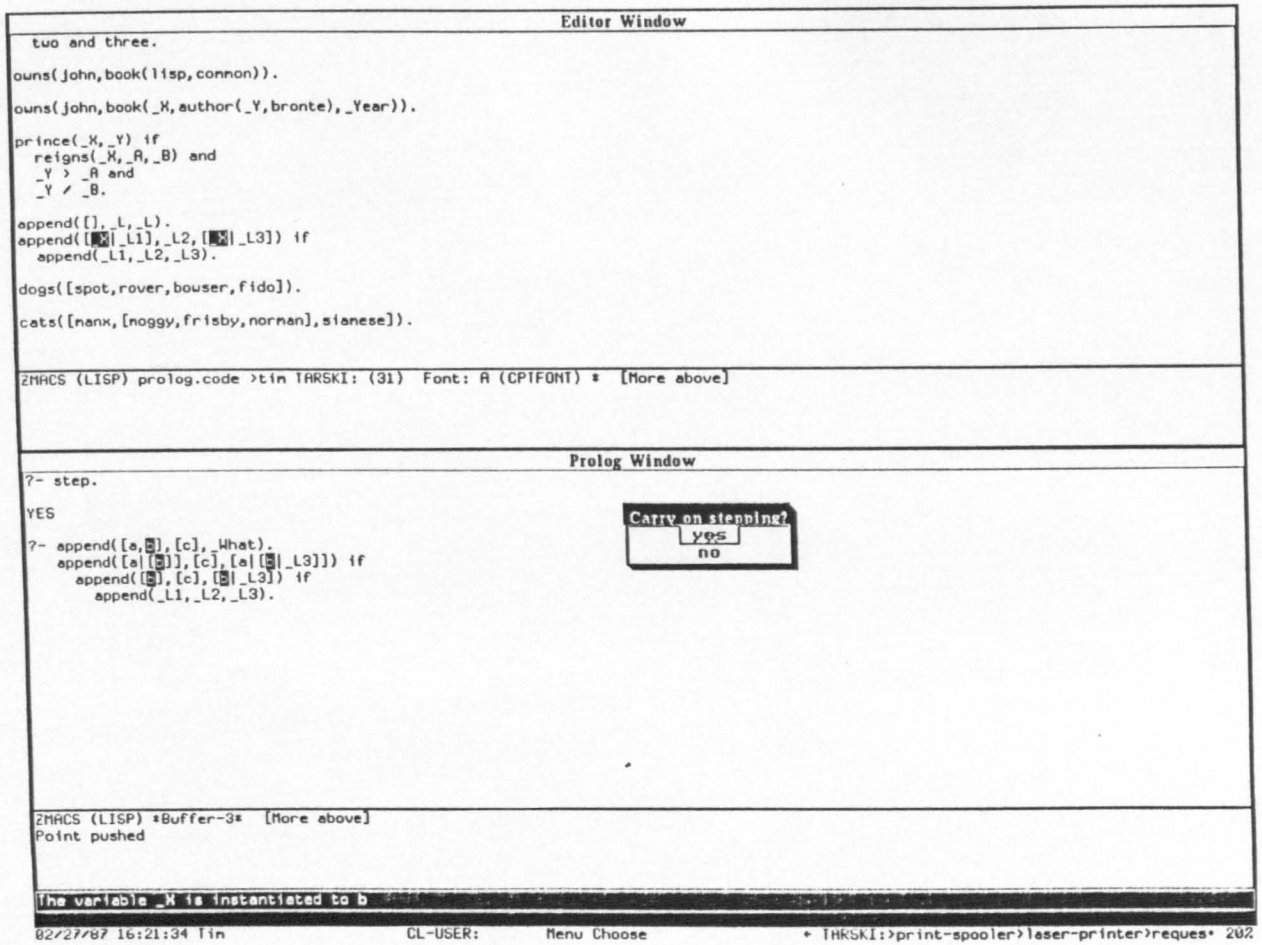

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([], L, L).
append(_X|_L1, _L2, _X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_L3]) if
append([b],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
About to instantiate the variables in the subgoal
02/27/87 16:19:02 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 762

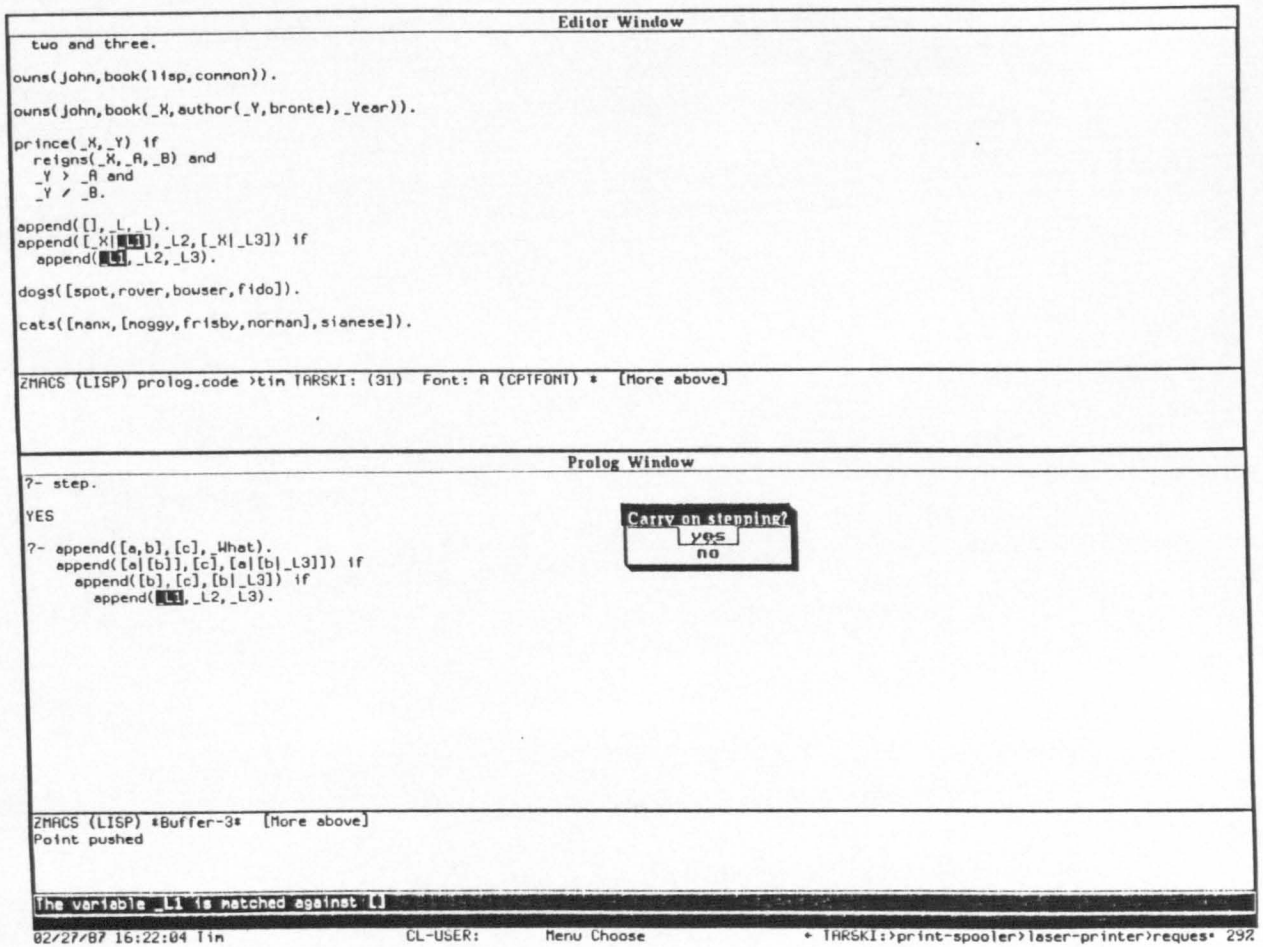

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,nornan],sianese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|_L3]) if
append([b],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L3 is matched against [_X|_L3]
02/27/87 16:19:08 Tin
CL-USER:
Menu Choose
+ TARSKI:>print-spooler>laser-printer>reques* 762

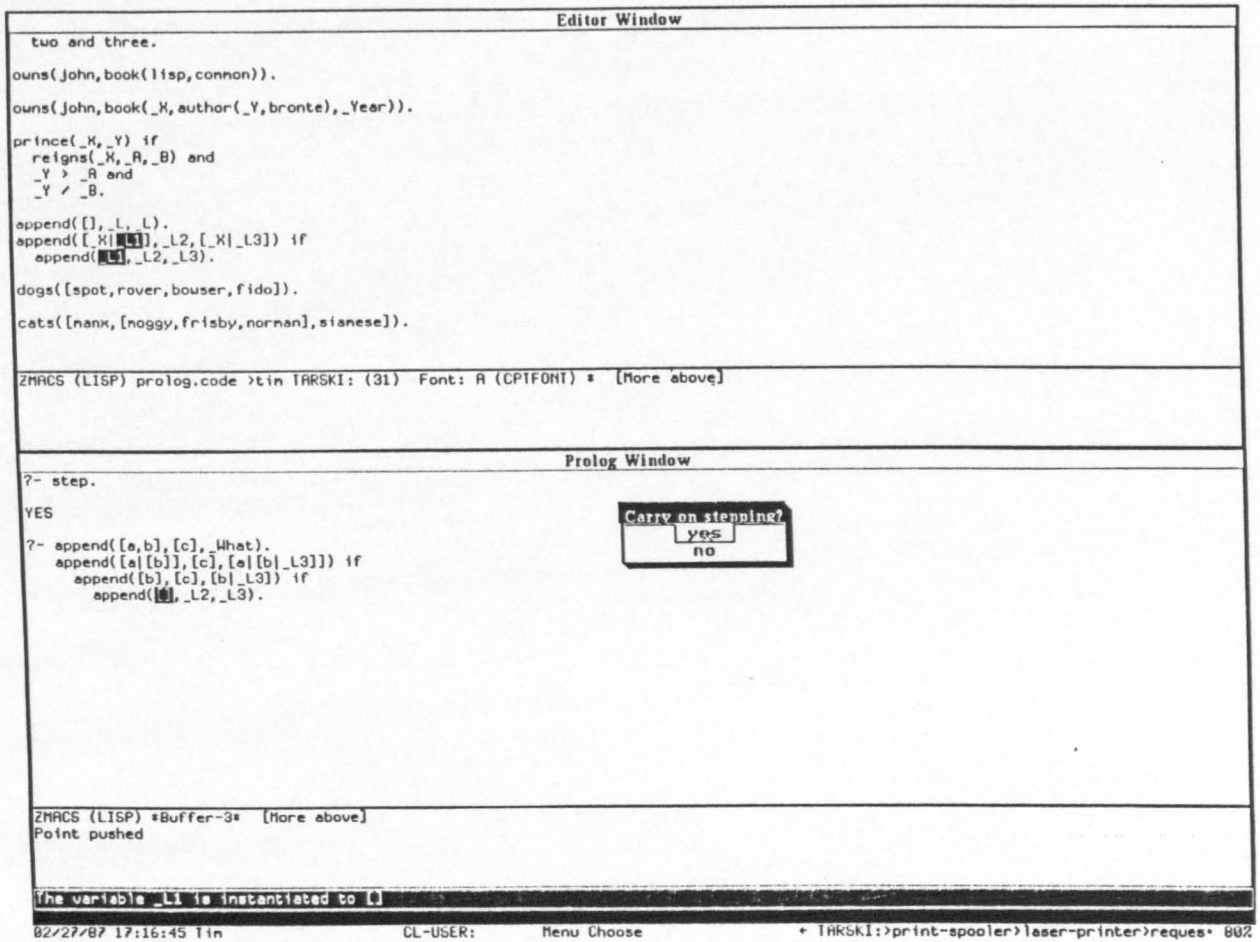

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([nanx,[noggy,frisby,norman],sianese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[_X|_L3]]) if
append([b],[c],[_X|_L3]).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L3 is instantiated to [_X|_L3]
02/27/87 16:19:18 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 762

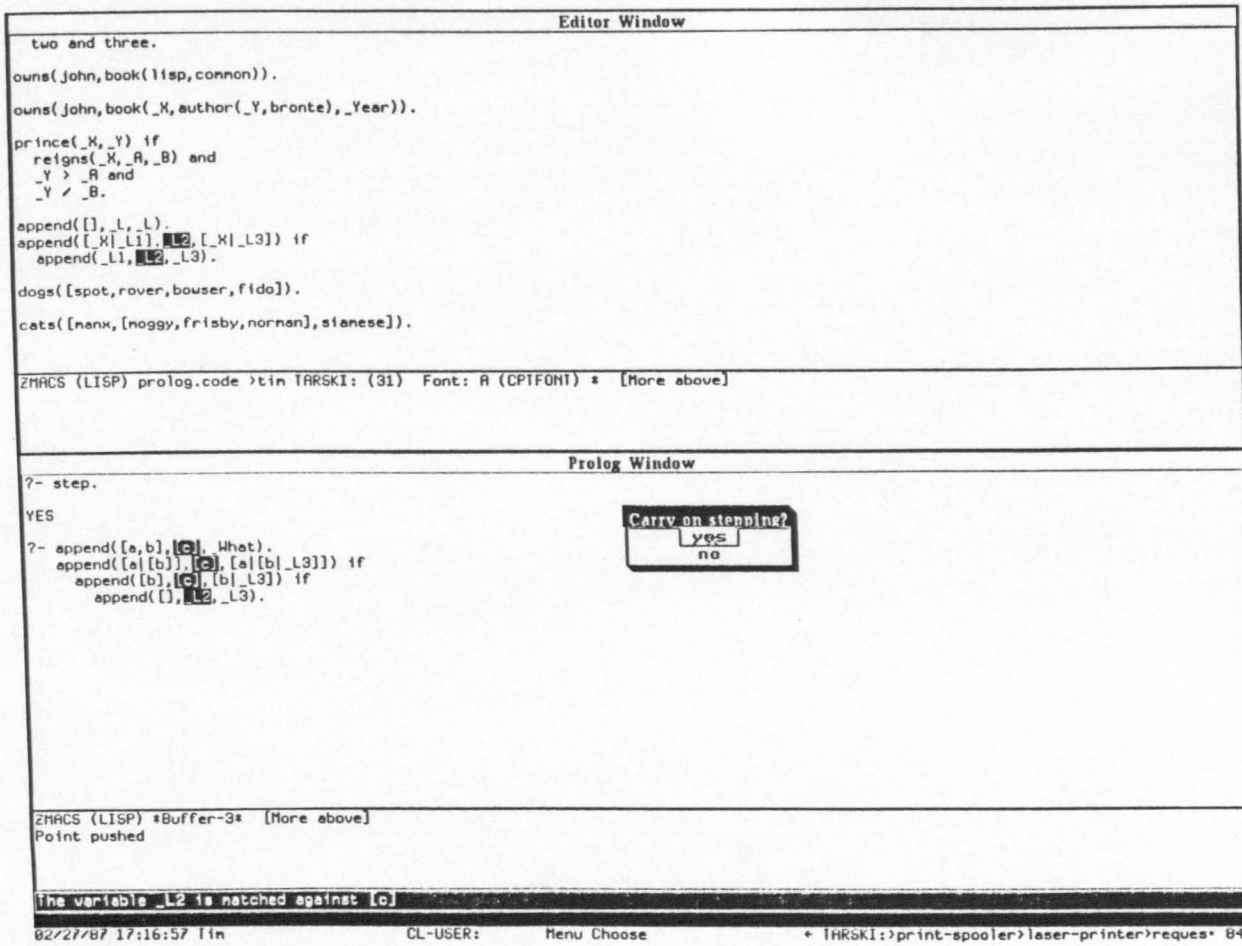

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[_X|_L3]]) if
append([b],[c],[_X|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Instantiating the subgoals of the rule whose head is append([_X|_L1],_L2,[_X|_L3])
02/27/87 16:21:10 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 14%

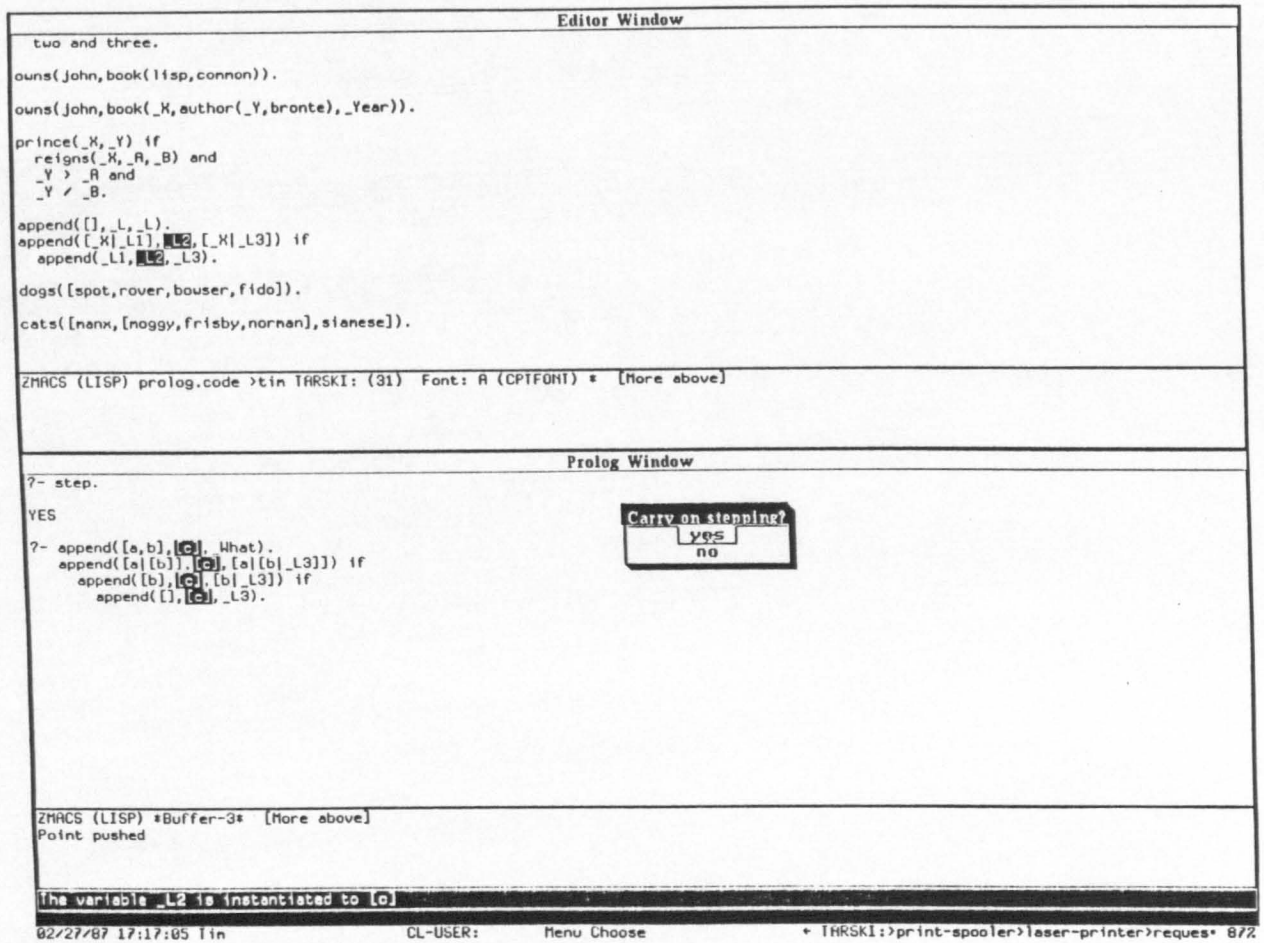

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A'(CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,_2],[c],_What).
append([a|[_3]],[c],[a|[_X|_L3]]) if
append([_3],[c],[_2|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _X is matched against b
02/27/87 16:21:27 Tin
CL-USER:
Menu Choose
TARSKI:>print-spooler>laser-printer>reques 192

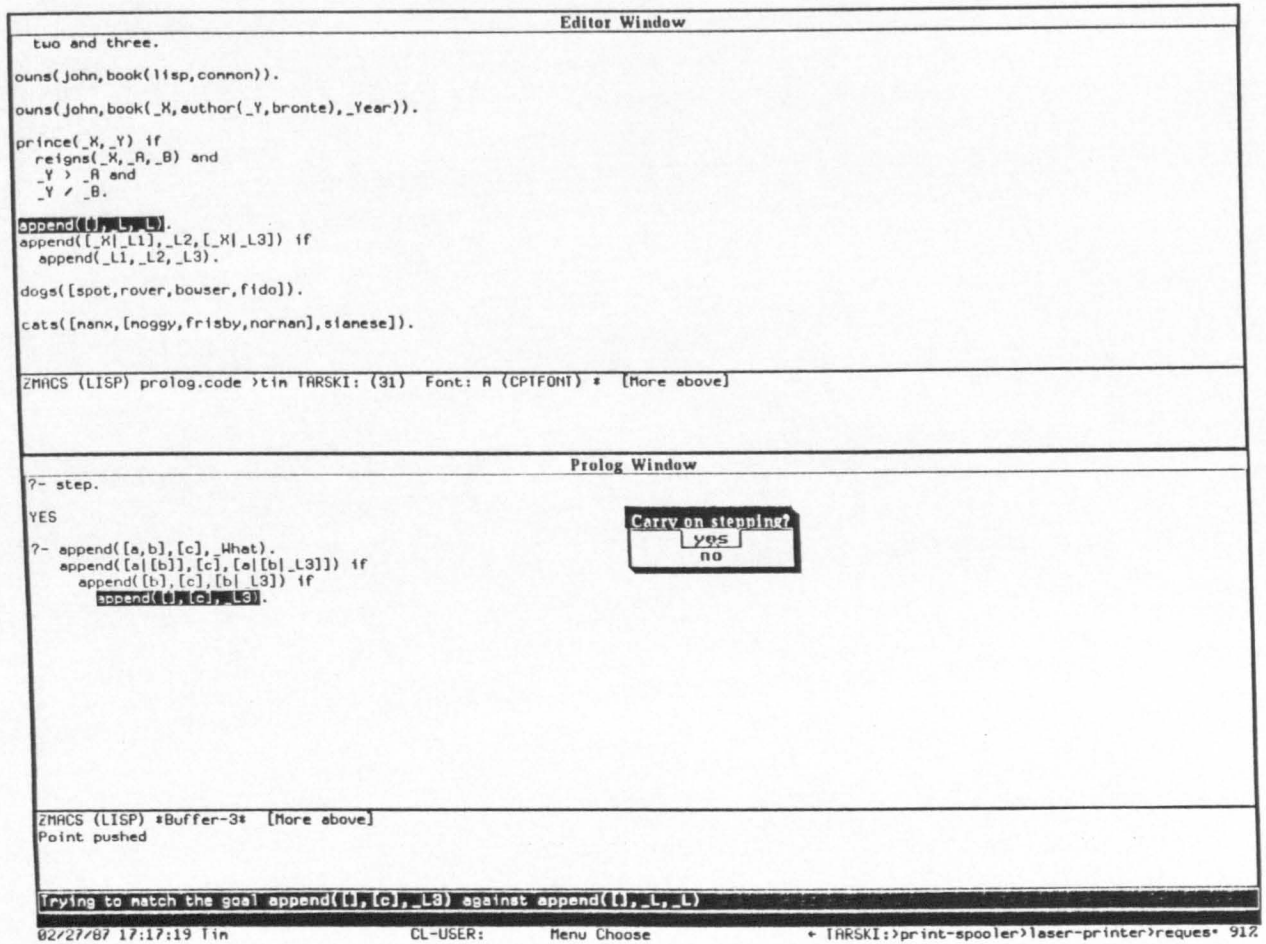

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([nanx,[noggy,frisby,norman],sianese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _X is instantiated to b
02/27/87 16:21:34 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 202

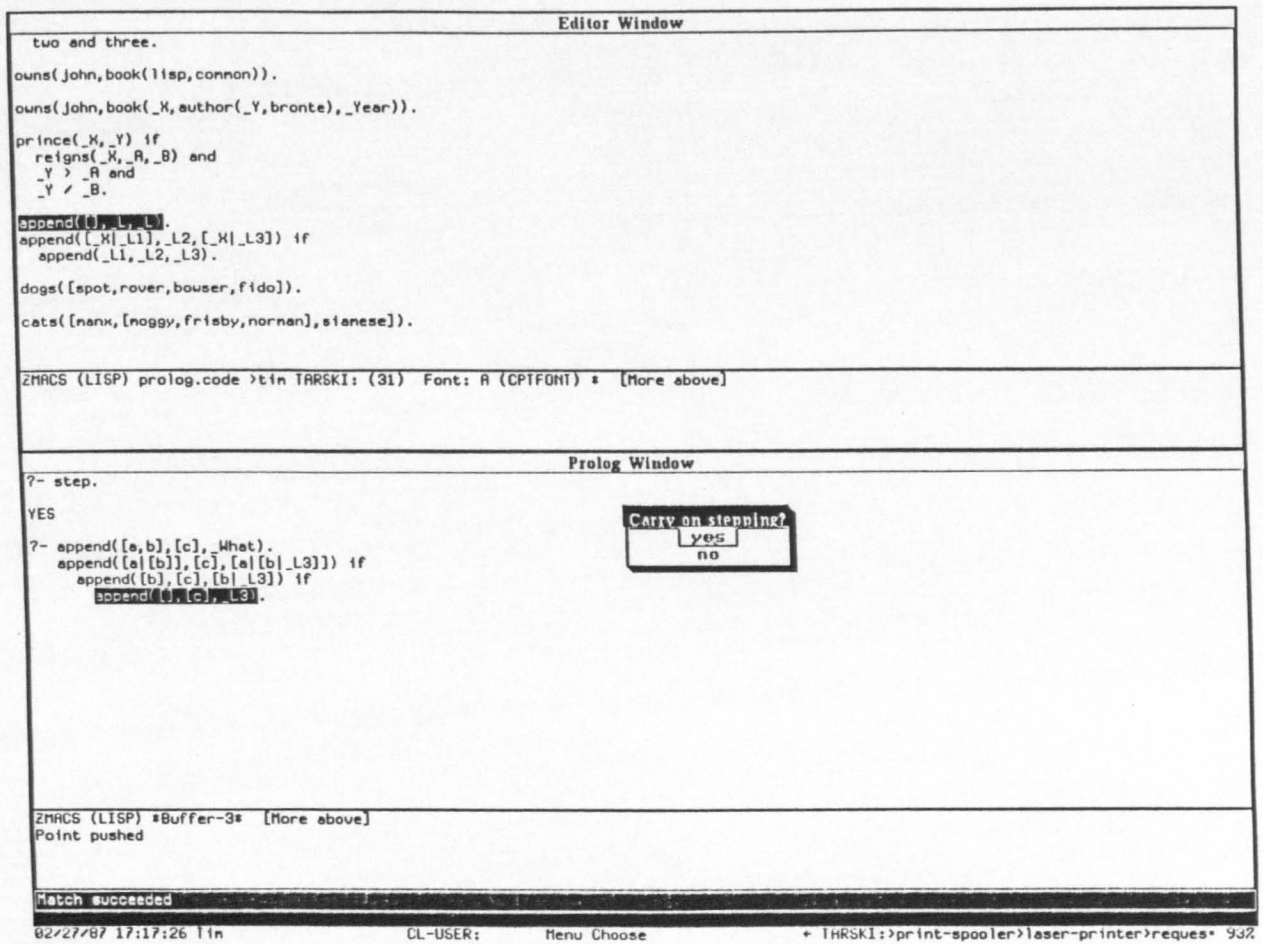

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append(_L1,_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L1 is matched against []
02/27/87 16:22:04 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 292

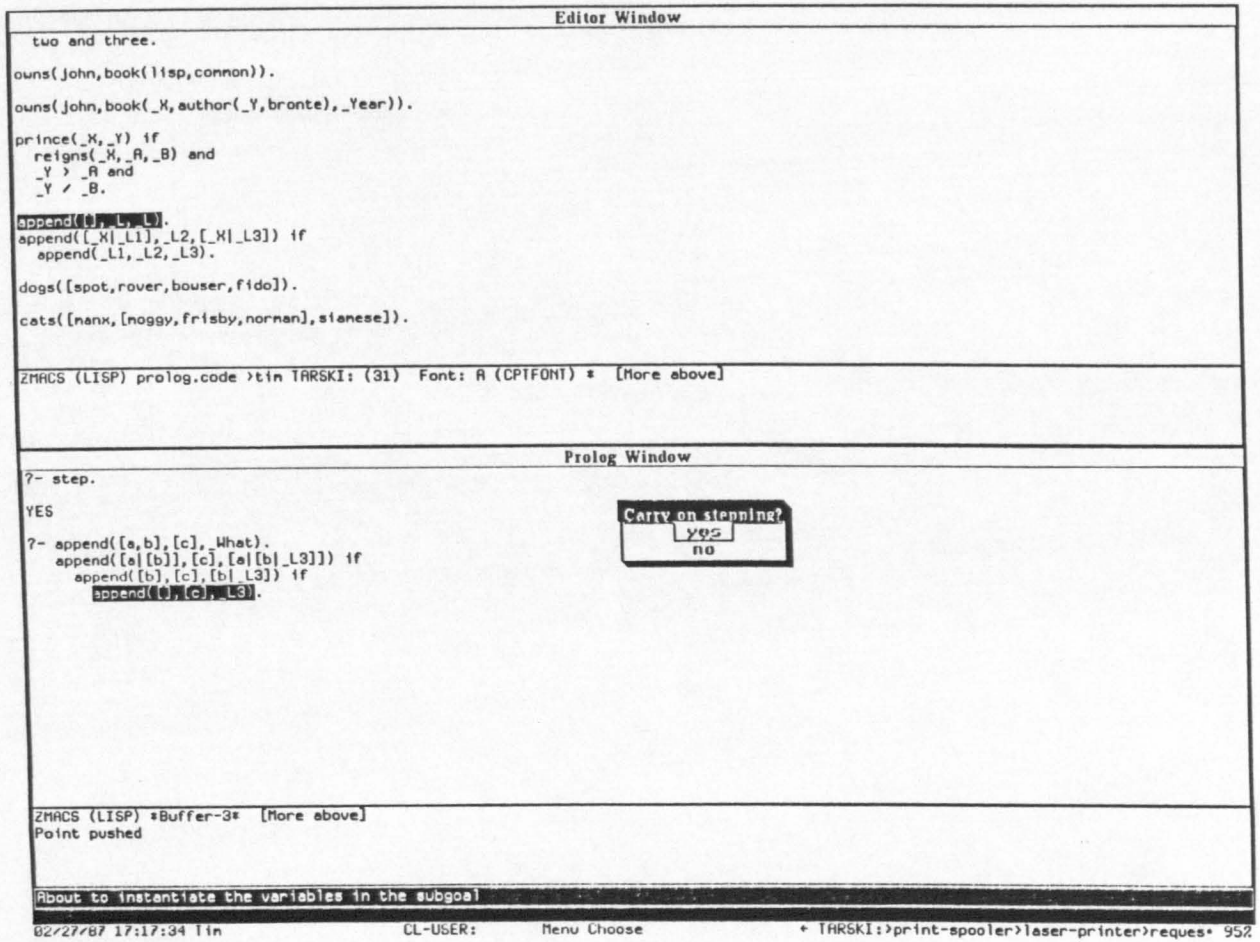

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append([],_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L1 is instantiated to []
02/27/87 17:16:45 Tim
CL-USER:
Menu Choose
TARSKI:>print-spooler>laser-printer>reques* 80%

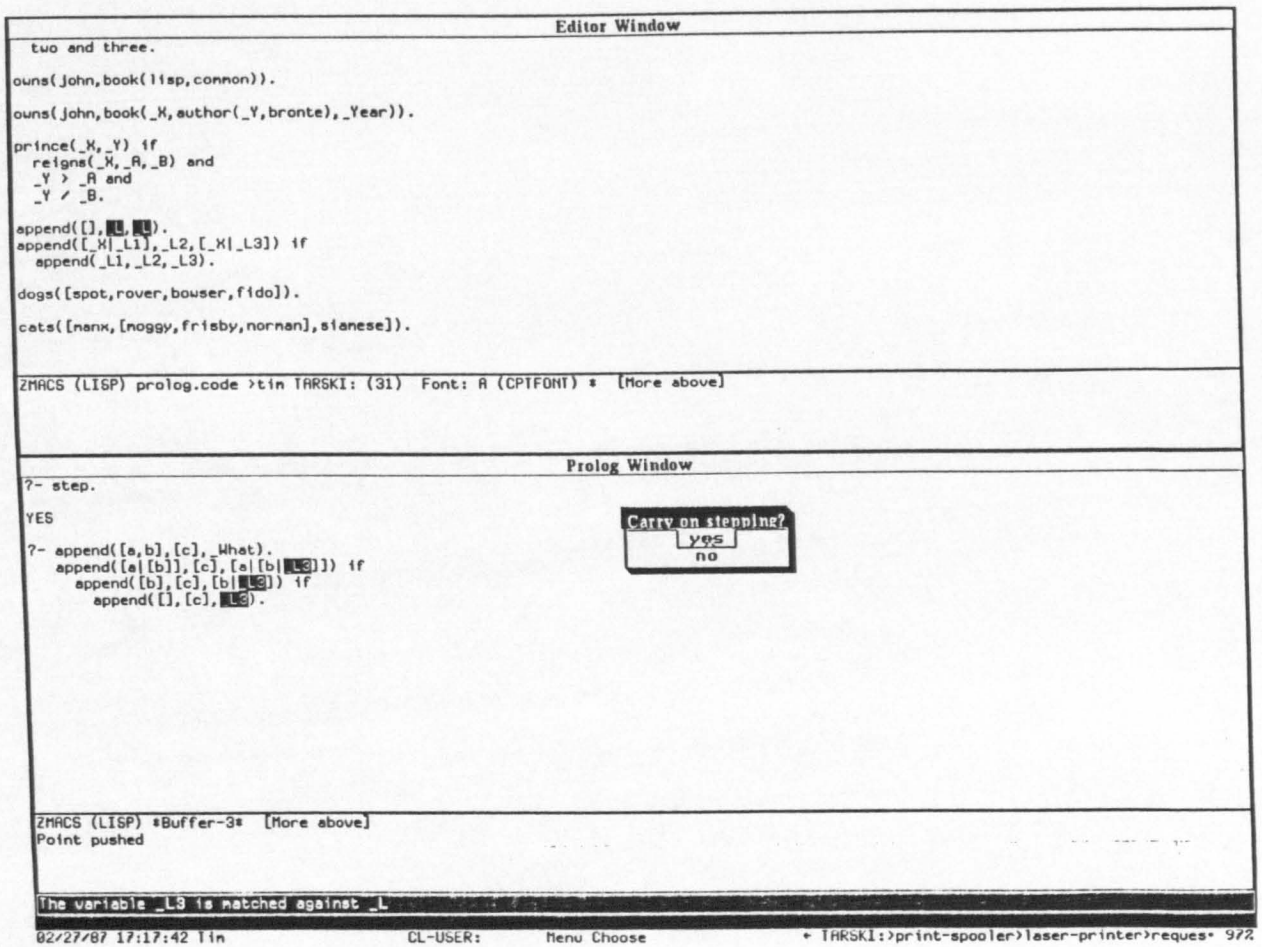

Editor Window
two and three.
owns(john,book(lisp,conmon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append([],_L2,_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
the variable _L2 is matched against [c]
02/27/87 17:16:57 tin
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 842

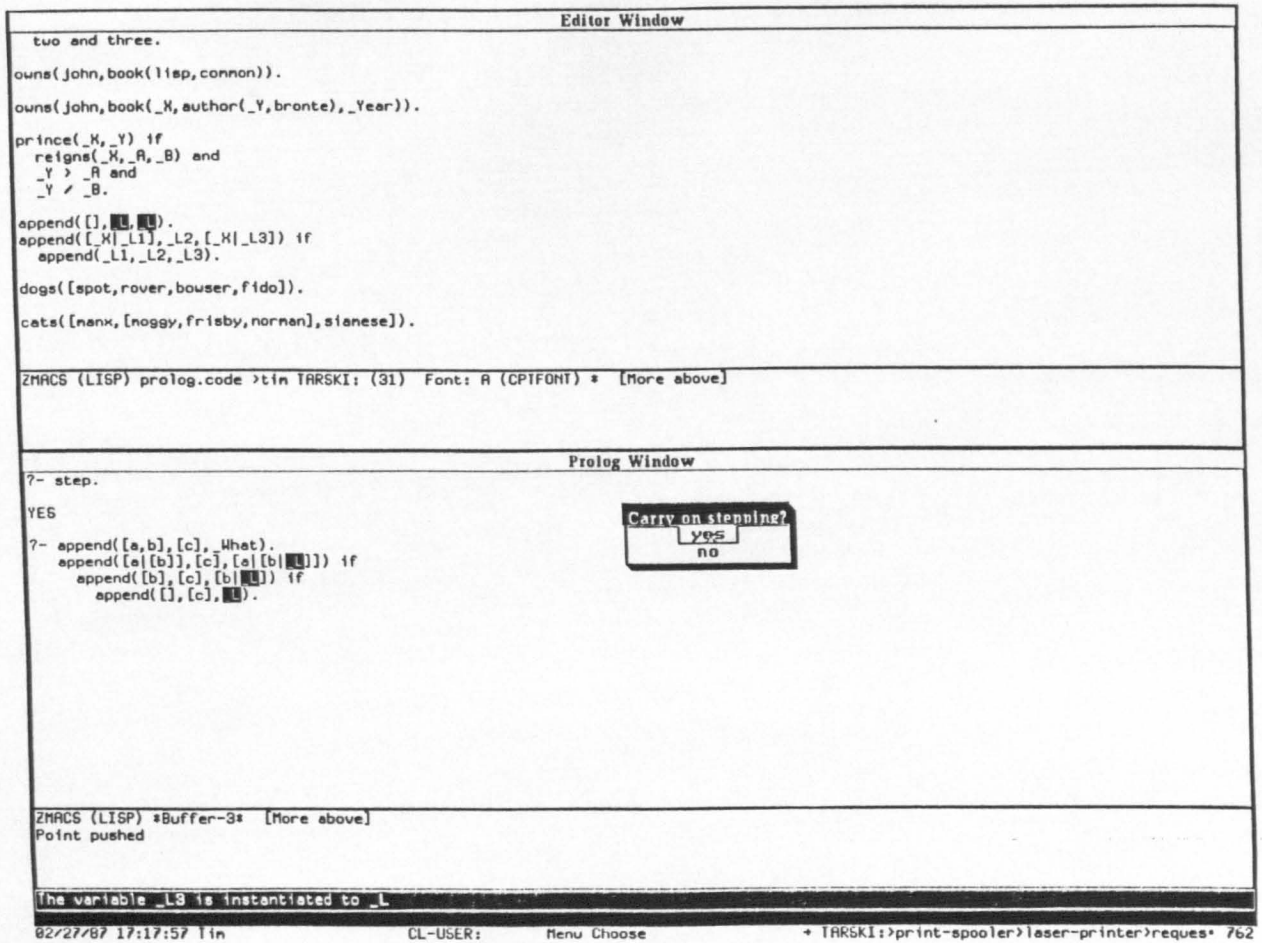

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append([],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L2 is instantiated to [c]
02/27/87 17:17:05 Tim
CL-USER:
Menu Choose
TARSKI:>print-spooler>laser-printer>reques* 872

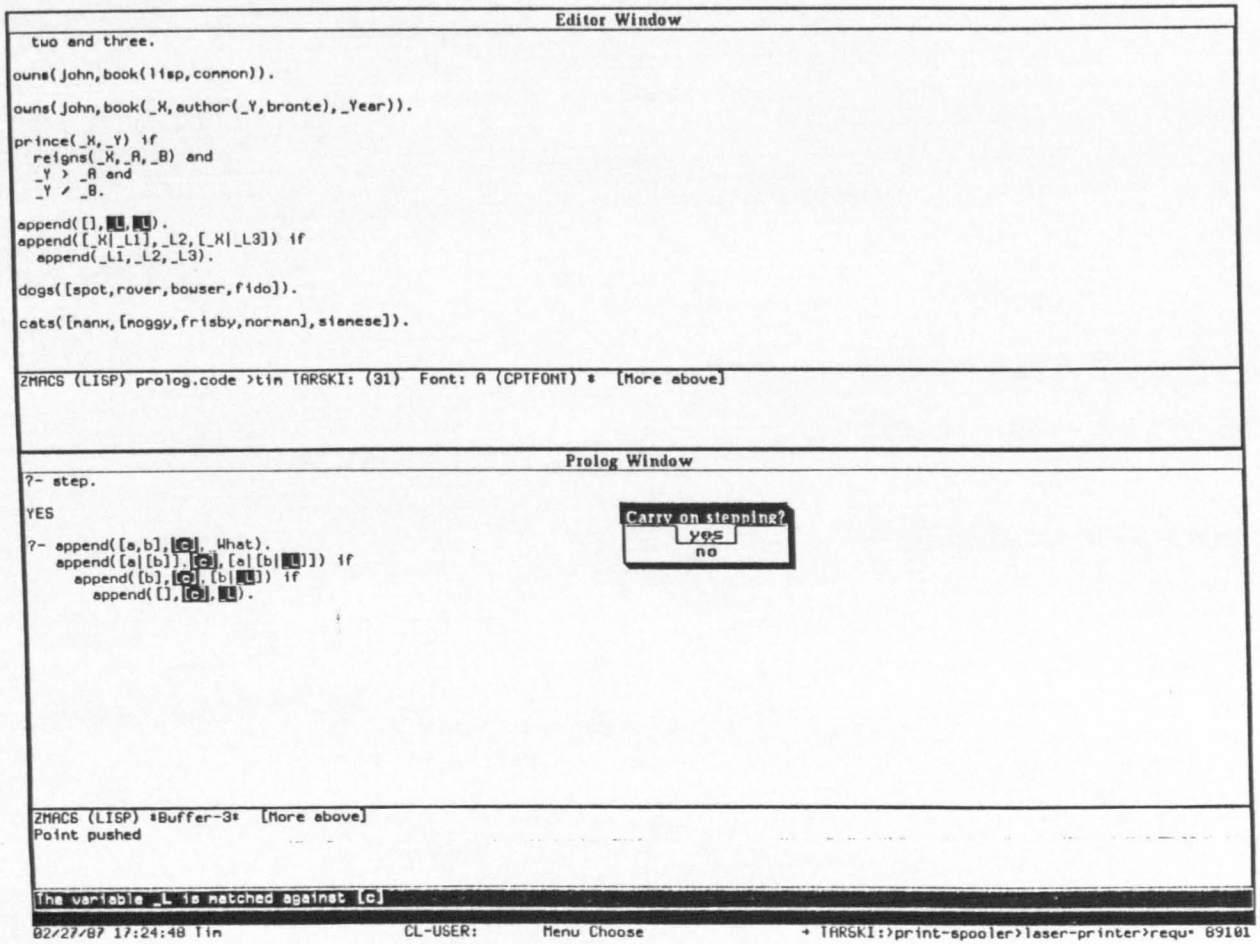

Editor Window
two and three.
ouns(john,book(lisp,connon)).
ouns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bouser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append([],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Trying to match the goal append([],[c],_L3) against append([],_L,_L)
02/27/87 17:17:19 Tim
CL-USER:
Menu Choose
TARSKI:>print-spooler>laser-printer>reques* 912

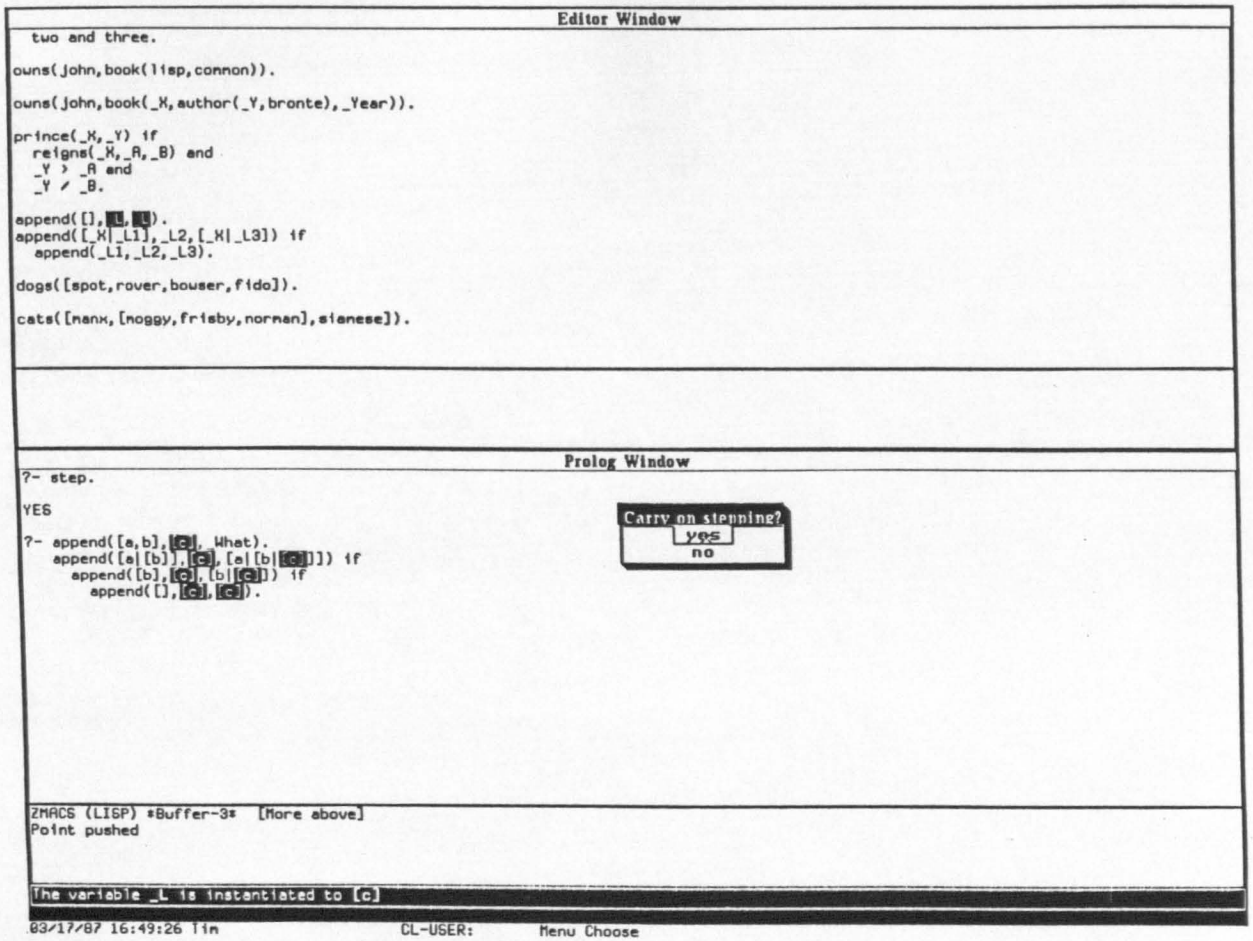

Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append([],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
Match succeeded
02/27/87 17:17:26 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 93%

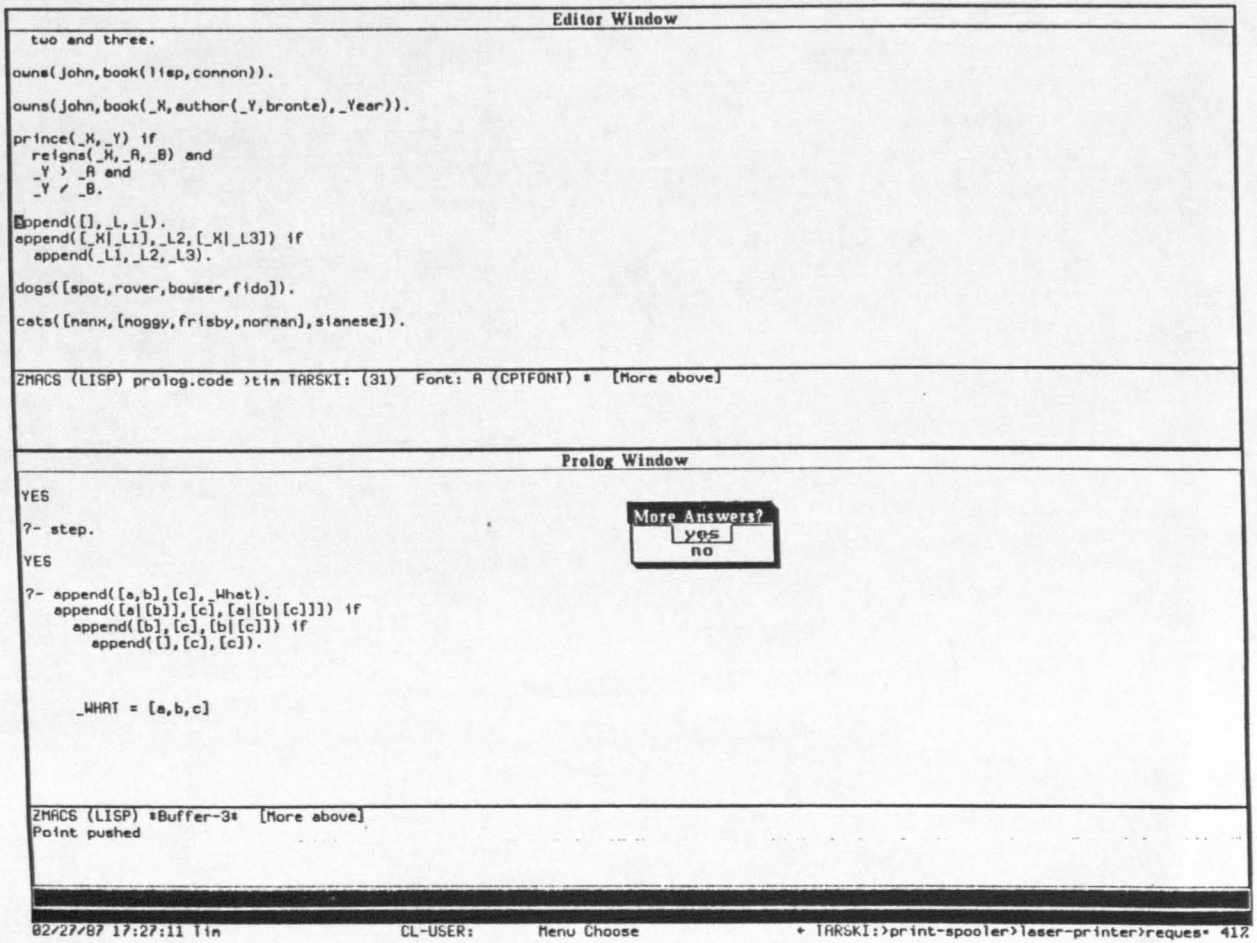

Editor Window
two and three.
owns(john,book(lisp,common)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L3]]) if
append([b],[c],[b|_L3]) if
append([],[c],_L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
About to instantiate the variables in the subgoal
02/27/87 17:17:34 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 952


Editor Window
two and three.
ouns(john,book(lisp,connon)).
ouns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],L,L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|L3]]) if
append([b],[c],[b|L3]) if
append([],[c],L3).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L3 is matched against _L
02/27/87 17:17:42 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 972


Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],sianese]).
ZMACS (LISP) prolog.code >tim TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|_L]]) if
append([b],[c],[b|_L]) if
append([],[c],_L).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
the variable _L3 is instantiated to _L
02/27/87 17:17:57 Tim
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 762


Editor Window
two and three.
owns(john,book(lisp,common)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],L,L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
?- step.
YES
?- append([a,b],[c],What).
append([a|[b]],[c],[a|[b|L]]) if
append([b],[c],[b|L]) if
append([],[c],L).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
the variable _L is matched against [c]
02/27/87 17:24:48 Tin
CL-USER:
Menu Choose
TARSKI:>print-spooler>laser-printer>requ* 89101


Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],L,L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([manx,[noggy,frisby,norman],siamese]).
Prolog Window
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|[c]]]) if
append([b],[c],[b|[c]]) if
append([],[c],[c]).
Carry on stepping?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
The variable _L is instantiated to [c]
03/17/87 16:49:26 Tim
CL-USER:
Menu Choose


Editor Window
two and three.
owns(john,book(lisp,connon)).
owns(john,book(_X,author(_Y,bronte),_Year)).
prince(_X,_Y) if
reigns(_X,_A,_B) and
_Y > _A and
_Y < _B.
append([],_L,_L).
append([_X|_L1],_L2,[_X|_L3]) if
append(_L1,_L2,_L3).
dogs([spot,rover,bowser,fido]).
cats([nanx,[noggy,frisby,norman],siamese]).
ZMACS (LISP) prolog.code >tin TARSKI: (31) Font: A (CPTFONT) * [More above]
Prolog Window
YES
?- step.
YES
?- append([a,b],[c],_What).
append([a|[b]],[c],[a|[b|[c]]]) if
append([b],[c],[b|[c]]) if
append([],[c],[c]).
_WHAT = [a,b,c]
More Answers?
yes
no
ZMACS (LISP) *Buffer-3* [More above]
Point pushed
02/27/87 17:27:11 Tin
CL-USER:
Menu Choose
* TARSKI:>print-spooler>laser-printer>reques* 412